



《编译原理》

第9讲：寄存器分配

何冬杰
重庆大学

Register Allocation Problem

- ❑ Given: a MIR program that uses an unbounded number of temporaries
 - e.g. the virtual registers in our MIR programs
- ❑ Find: a mapping from temporaries to machine registers such that
 - program semantics is preserved (i.e., behavior is the same)
 - register usage is maximized
 - moves between registers are minimized
 - calling conventions/architecture requirements are obeyed
- ❑ Register allocation is generally an **NP-complete** problem
 - Can we allocate all n temporaries to k registers?
- ❑ But we have heuristics that are linear in practice!
- ❑ **Stack Spilling**
 - If there are k registers available and $m > k$ temporaries are live at the same time, then not all of them will fit into registers
 - So, "**spill**" the excess temporaries to the stack.

Naïve Register Allocation

❑ Spill all registers into the stack (see [SpillLowering.ml](#))

- **安排栈槽**: 给每个虚拟寄存器分配一个独立的栈槽

- 扫描整个函数，为收集到的虚拟寄存器分配栈槽
- 同一个虚拟寄存器始终使用同一个槽，不同虚拟寄存器不共享槽
- 栈槽编号由扫描收集的顺序决定

- **落实栈帧**: 计算栈槽偏移，把抽象的栈槽地址转换成相对fp的地址，并生成栈帧元信息

- **改写指令**: 使用前加载，计算后写回

- 每条指令执行前，用 t0、t1、t2 将需要的值从栈中加载出来
- 执行计算，再把结果写回对应栈槽
- 下一条指令重新加载，不跨指令保留这些临时寄存器中的值

❑ Example:

```
sub %v6, %v3, %v5
```



```
ld  %t0, 48(%fp)
ld  %t1, 40(%fp)
sub %t2, %t0, %t1
sd  32(%fp), %t2
```

❑ Can we do better?

Linear-Scan Register Allocation

□ Simple, greedy register-allocation strategy:

- Compute liveness information: **live_out(x)**
 - **live_out(x)** is the set of virtual registers that are live immediately after x's definition
- Let **palette** be the set of usable registers
 - Usually reserve two physical registers (t0, t1) as scratch registers for spill code
 - **palette** = {t2-t6, s1-s11}, t2-t6 are caller-saved temporary registers, s1-s11 are callee-saved
- Maintain a "layout" **loc_map** that maps virtual registers to locations
 - locations include **physical registers** and **stack slots** n, starting at n=0
- Scan through the program. For each instruction that defines a virtual register **x**

```
used = {r | reg r = loc_map(y) s.t. y ∈ live_out(x)}  
available = palette - used  
If available is empty:      // no registers available, spill  
    loc_map(x) := slot n; n = n + 1  
Else:  
    pick r in available:    // choose an available register  
    loc_map(x) := reg r
```

Spill when not available.

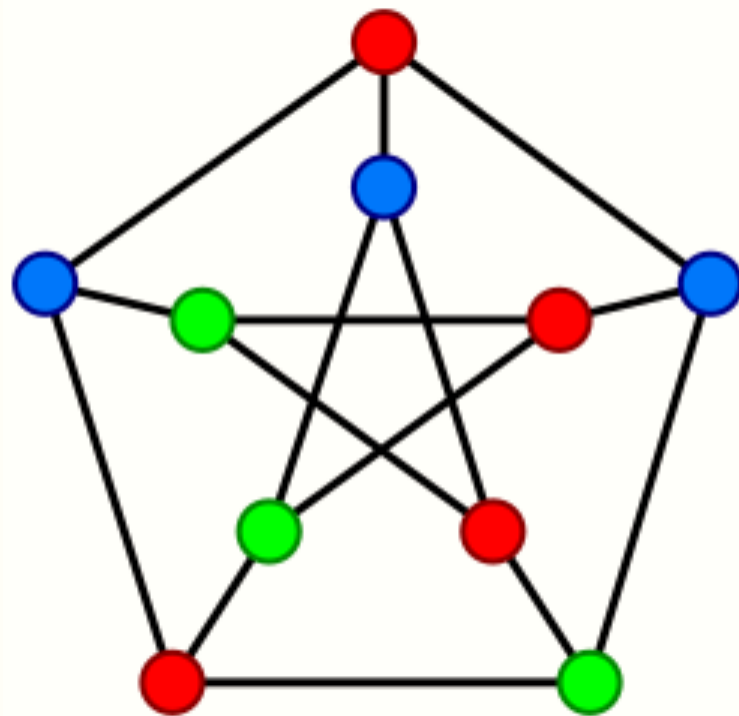
□ **Exercise:** greedy_layout: assign registers greedily using linear scan

- Can we do better?

Graph-Coloring-based Register Allocation

□ Graph coloring

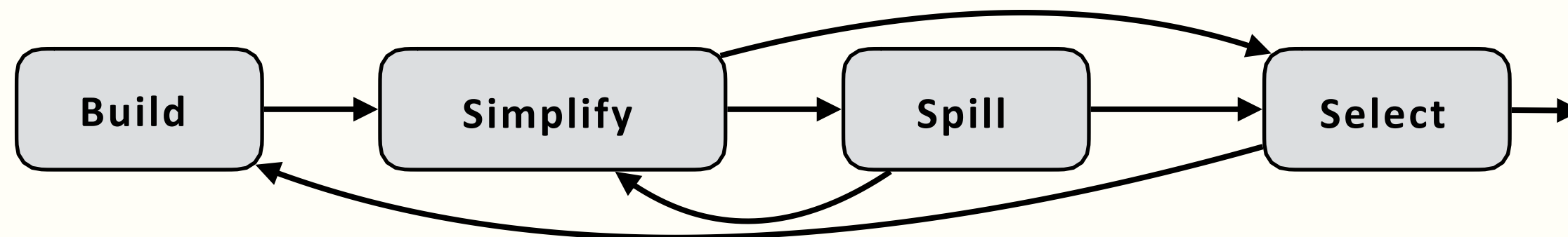
- Given a graph, can we assign one of k colors to each node such that connected nodes have different colors?
- Here, nodes are temp variables, an edge between t_1 and t_2 means that t_1 and t_2 are live at the same time. Colors are registers.



□ But graph coloring is also **NP-complete**! How does that work?

Coloring by Simplification

- ❑ **Build**: construct an interference graph, using dataflow analysis to find for each program point, variables that are live at the same time
- ❑ **Simplify**: color based on a simple heuristic
 - If graph G has node n with $k - 1$ edges, then $G - \{n\}$ is k -**colorable** iff G is k -colorable
 - So remove nodes with degree $< k$
- ❑ **Spill**: if the graph has only nodes with a degree $\geq k$, choose one to **potentially spill**
 - i.e., that **may** need to be saved to the stack
 - Then continue with Simplify
- ❑ **Select**: when the graph is empty, start restoring nodes in reverse order and color them
 - When we encounter a potential spill node, try coloring it. If we can't, rewrite the program to store it to the stack after definition and load before use. Try again!

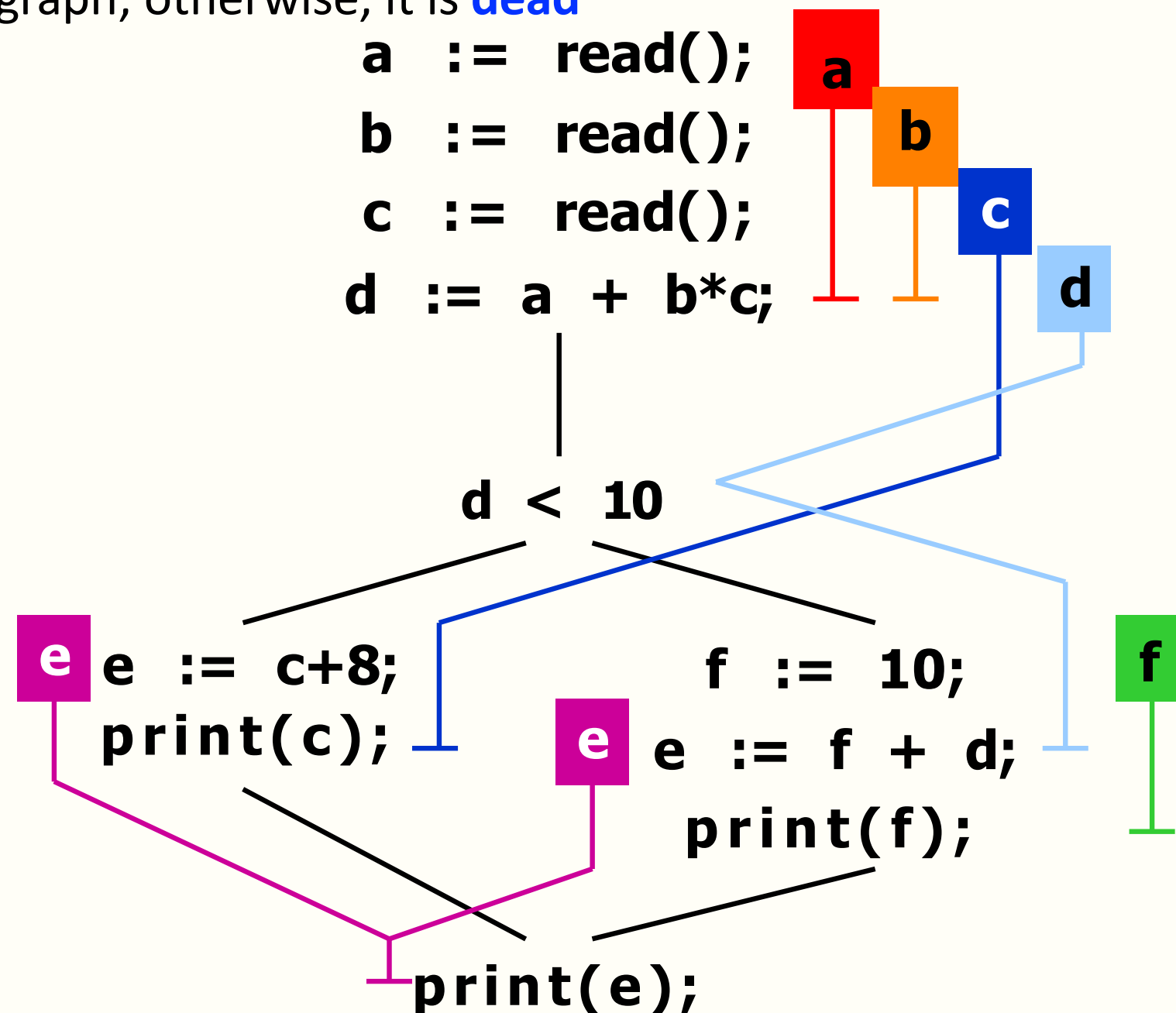


Build Interference Graph

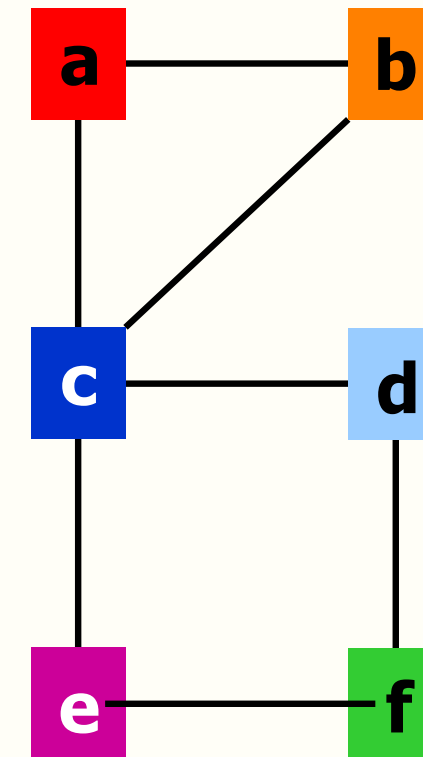
❑ **Procedure-wide register allocation:** only live variables require register storage

❑ **Live Variable Analysis** (a kind of dataflow analysis)

- A variable is **live** at node **N** if the value it holds is used on some path further down the control-flow graph; otherwise, it is **dead**



Interference Graph



Two variables(values) interfere when their live ranges overlap

Coloring by Simplification: Example

{live-in: j, k}

`g := * (j+12)`

`h := k - 1`

`f := g * h`

`e := * (j+8)`

`m := * (j+16)`

`b := * (f+0)`

`c := e + 8`

`d := c`

`k := m + 4`

`j := b`

{live-out: d, j, k}

**Live variables at
each program point**

{j, k}

{g, j, k}

{g, h, j}

{f, j}

{e, f, j}

{e, f, m}

{b, e, m}

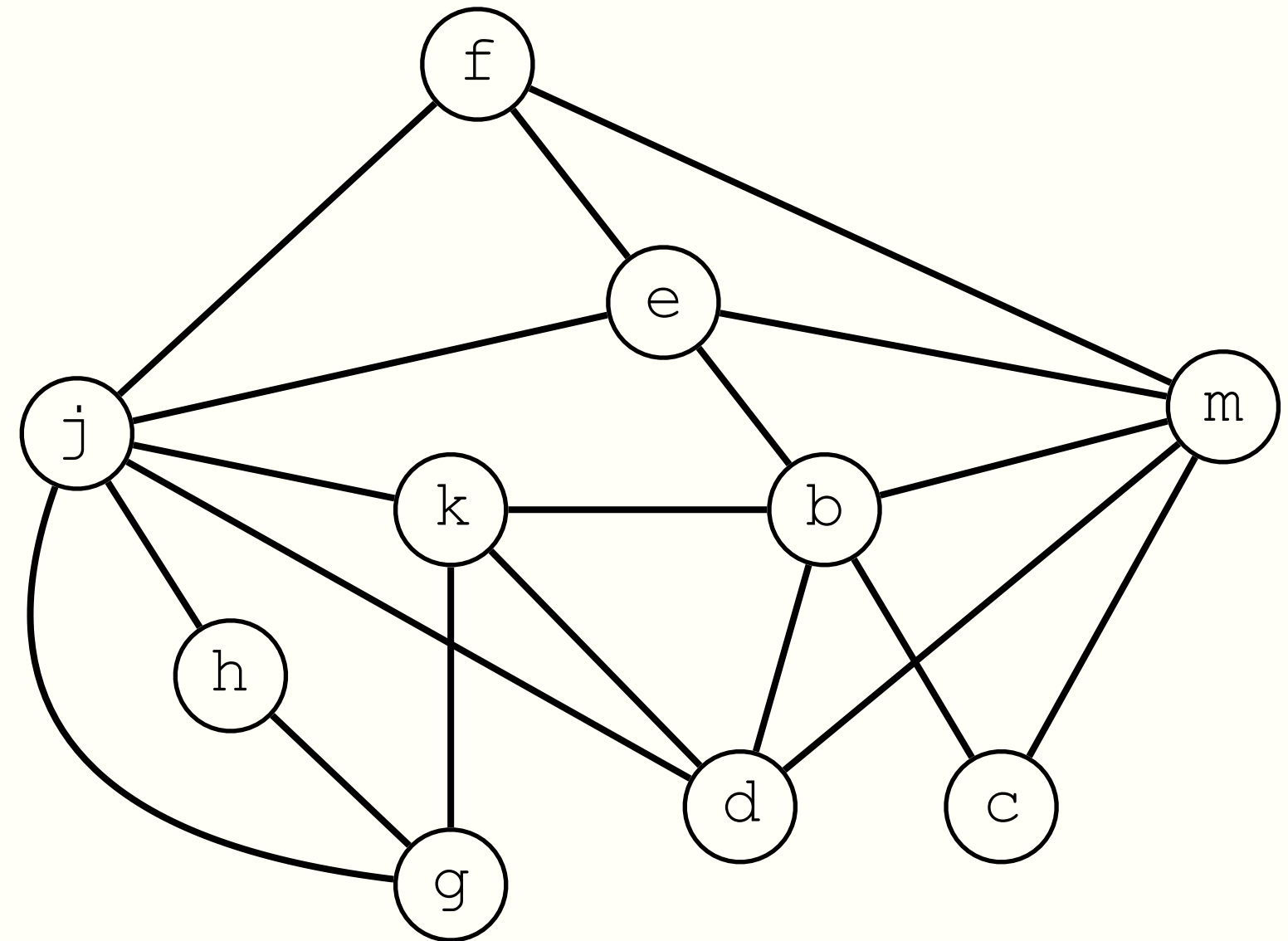
{c, b, m}

{b, d, m}

{b, d, k}



Interference graph



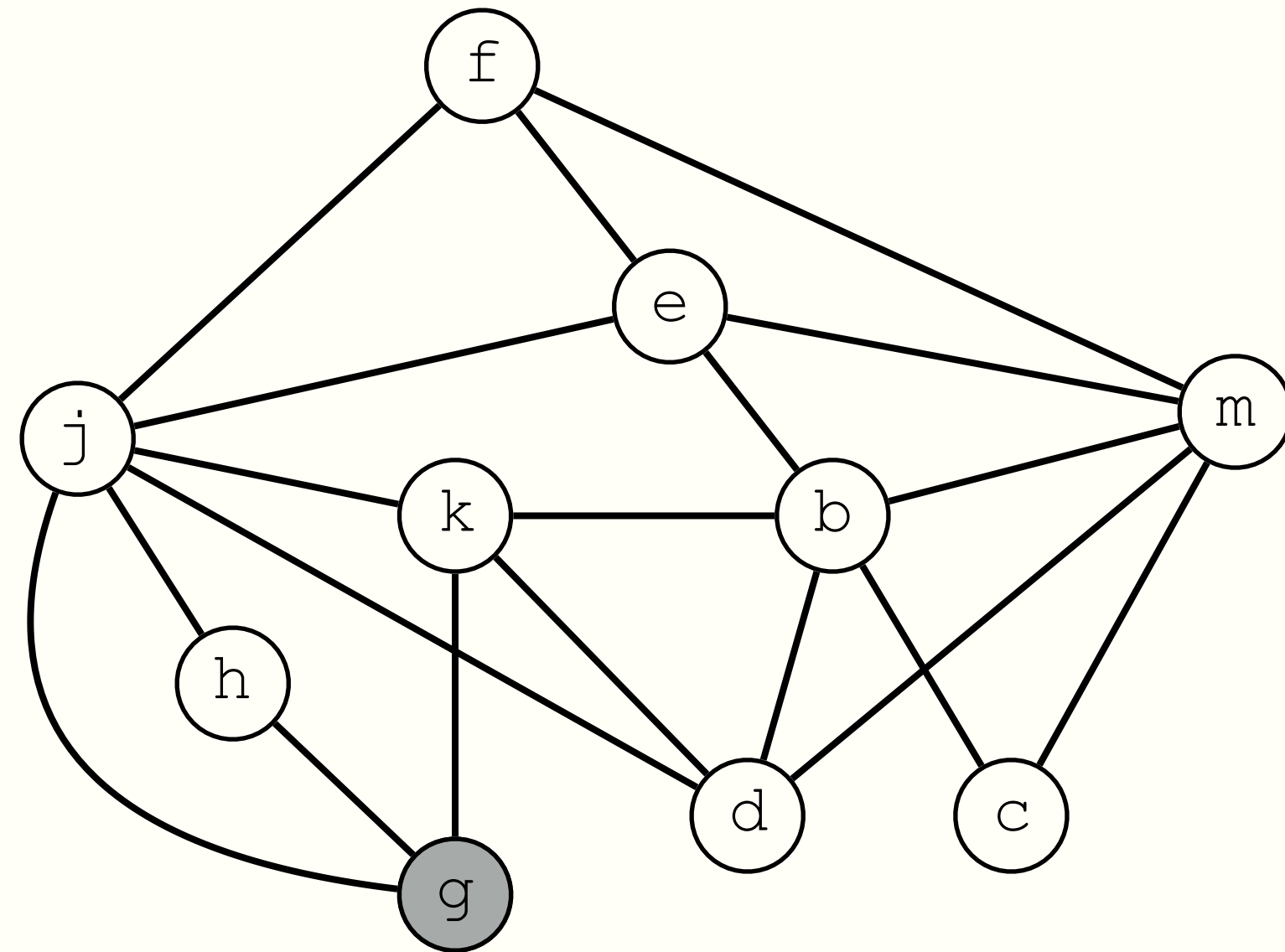
An **edge** (a, b) is added if variables a and b may be **live simultaneously at some program point**.

Simplification (4 registers)

Choose any node with degree < 4

Stack:

g



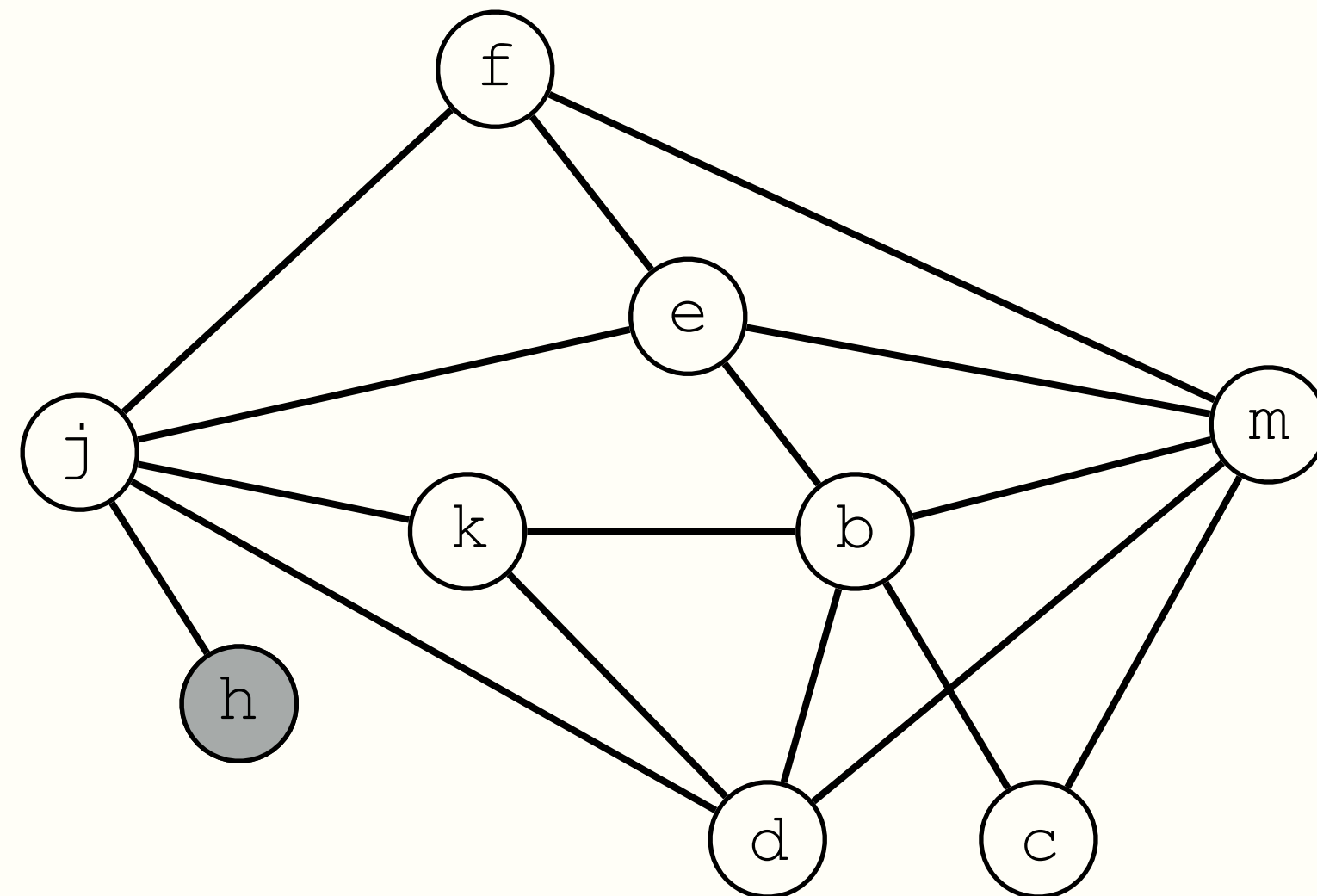
Simplification (4 registers)

Choose any node with degree < 4

Stack:

g

h

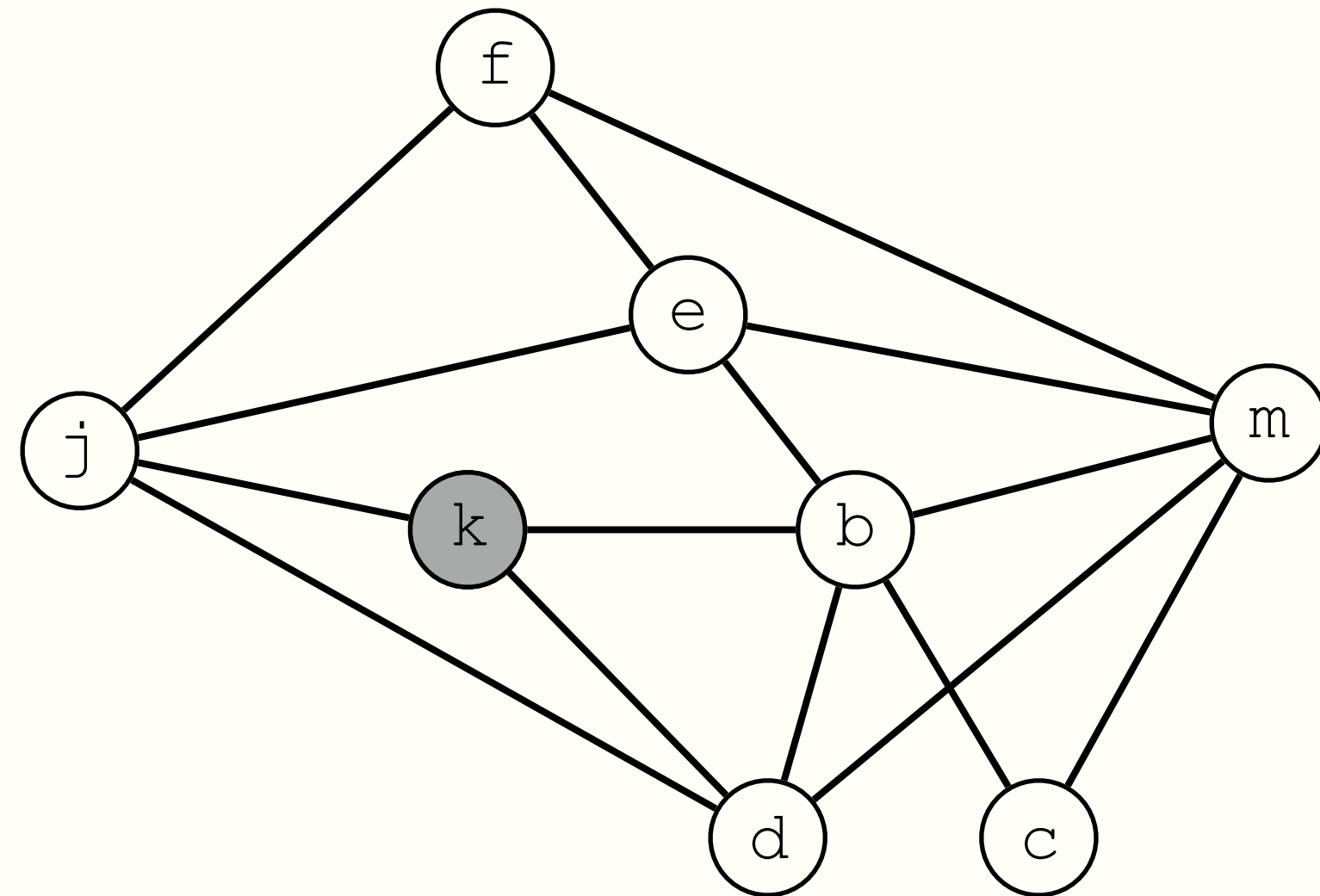


Simplification (4 registers)

Choose any node with degree < 4

Stack:

g
h
k

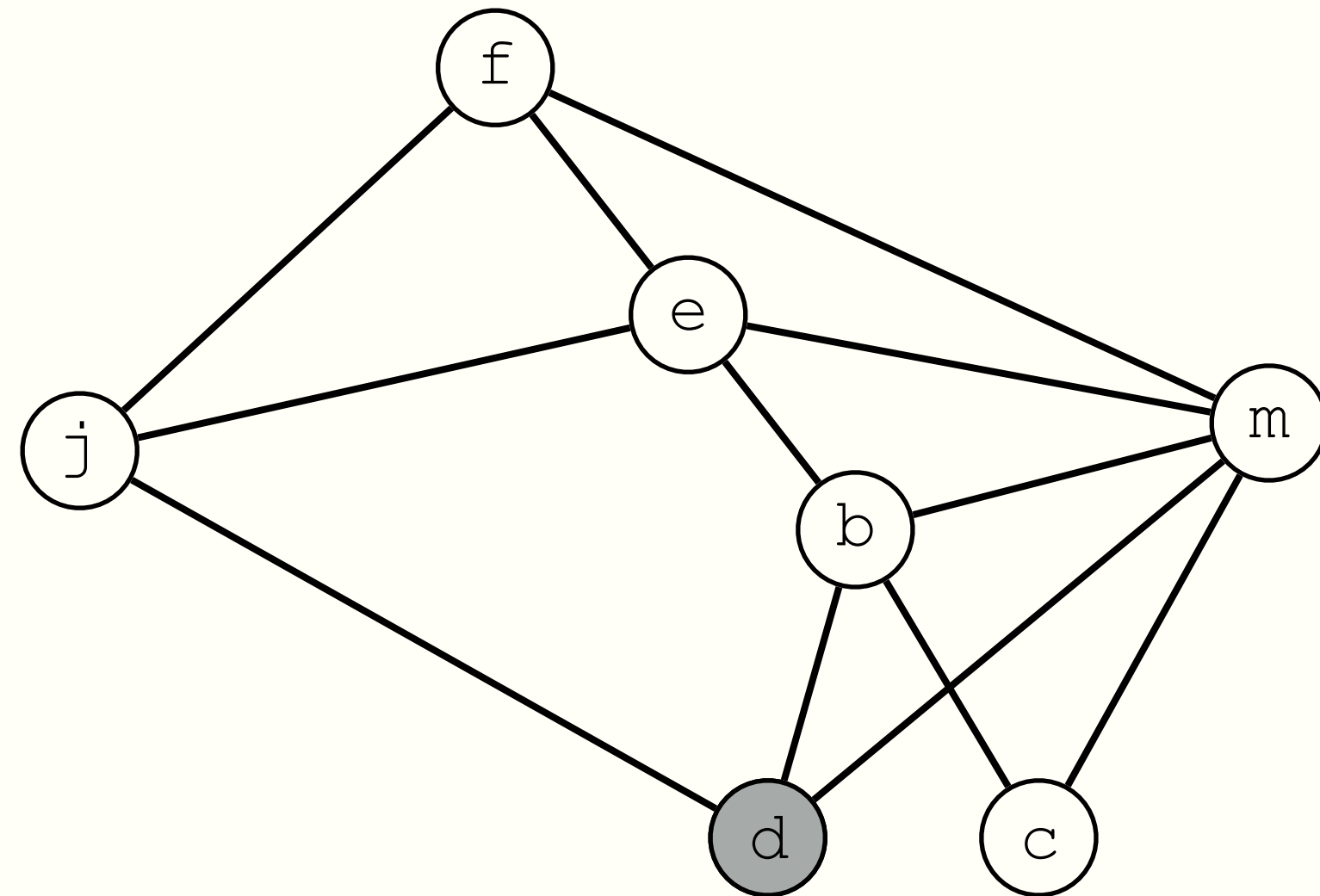


Simplification (4 registers)

Choose any node with degree < 4

Stack:

g
h
k
d

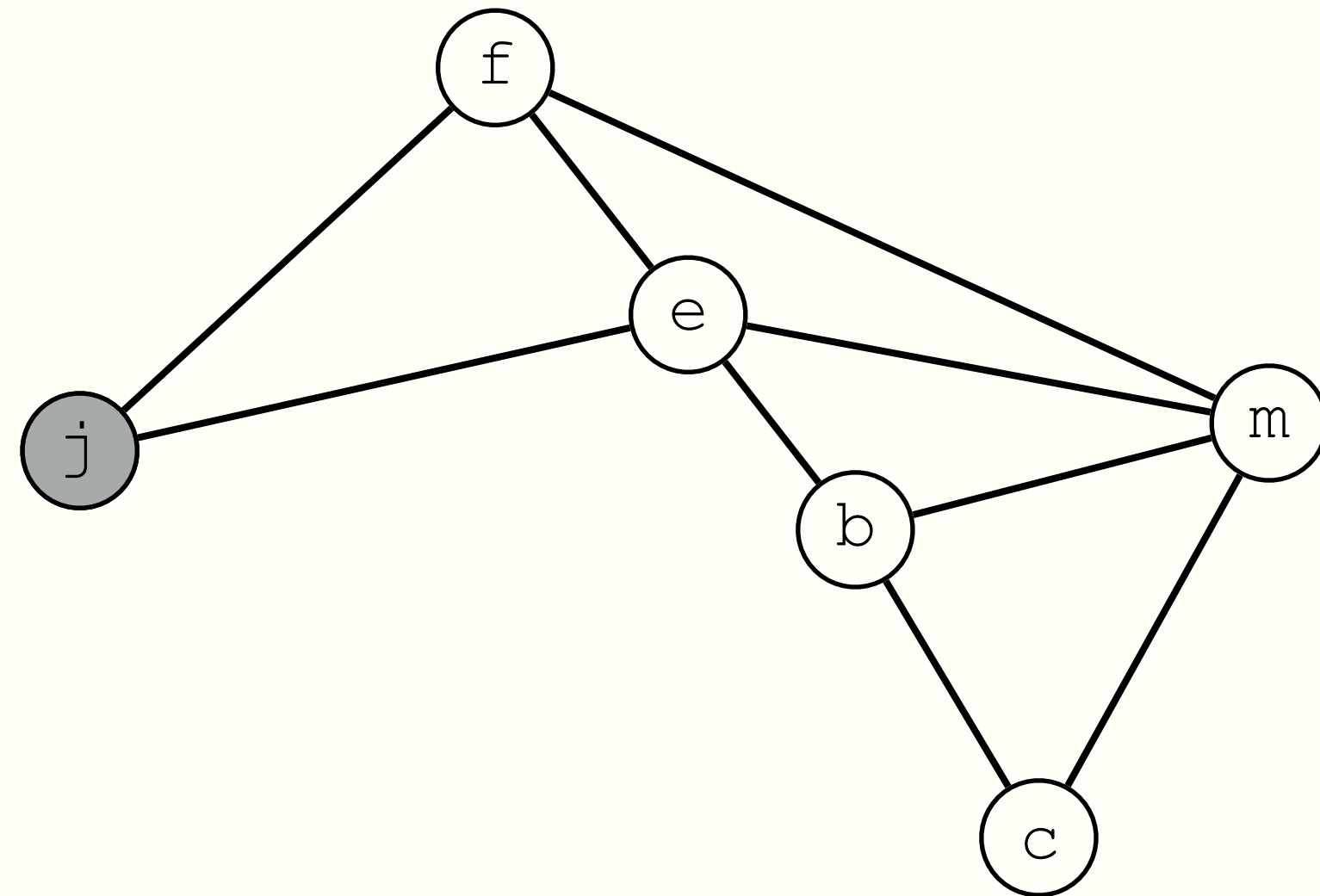


Simplification (4 registers)

Choose any node with degree < 4

Stack:

g
h
k
d
j



Simplification (4 registers)

Choose any node with degree < 4

Stack:

g

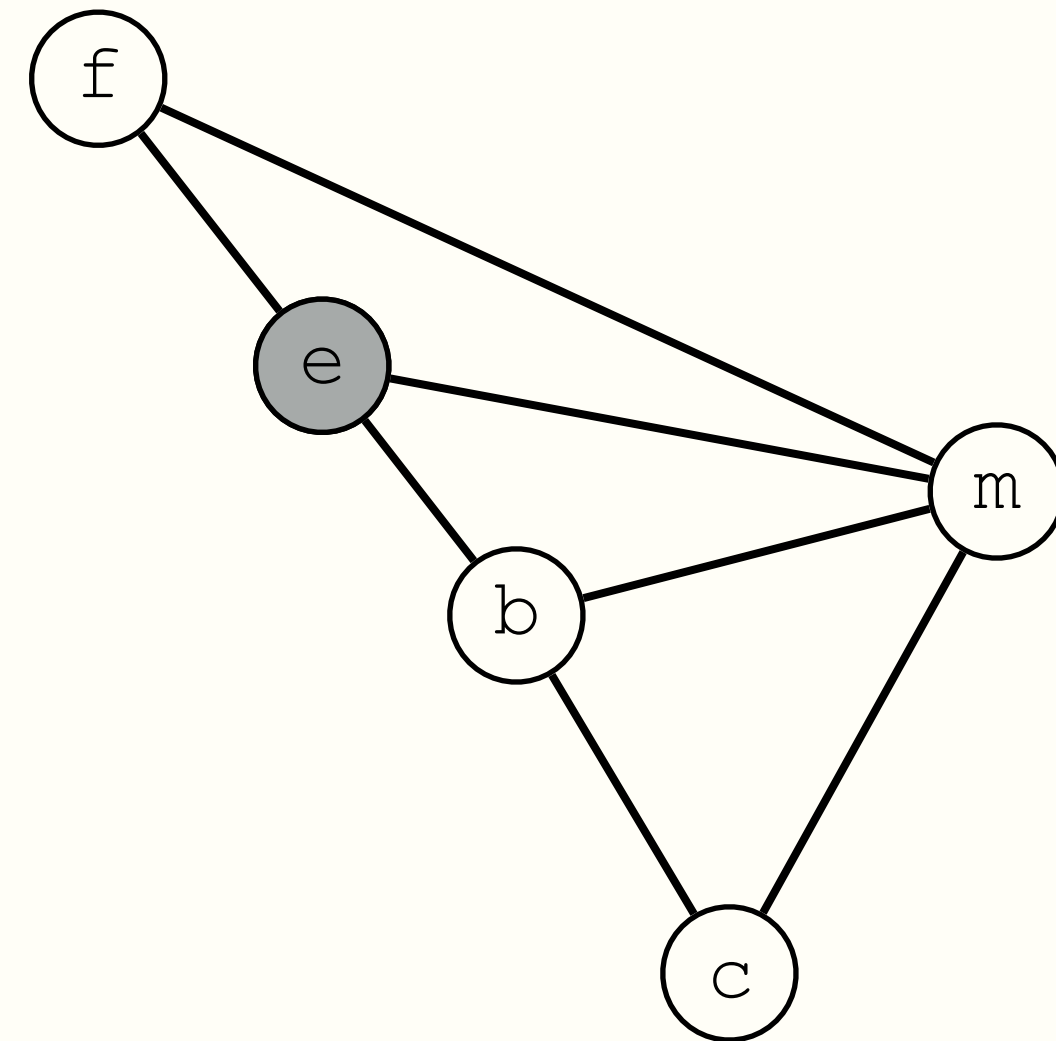
h

k

d

j

e



Simplification (4 registers)

Choose any node with degree < 4

Stack:

g

h

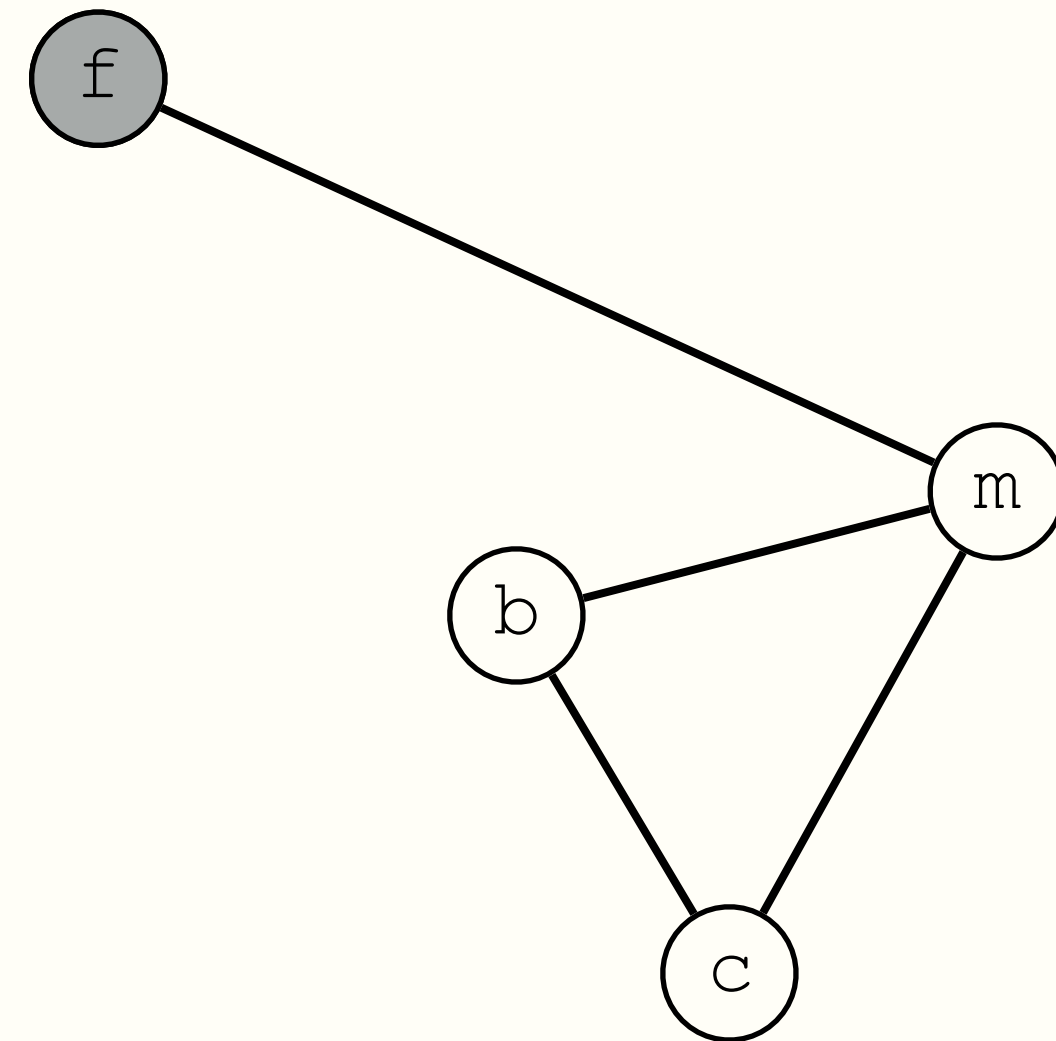
k

d

j

e

f



Simplification (4 registers)

Choose any node with degree < 4

Stack:

g

h

k

d

j

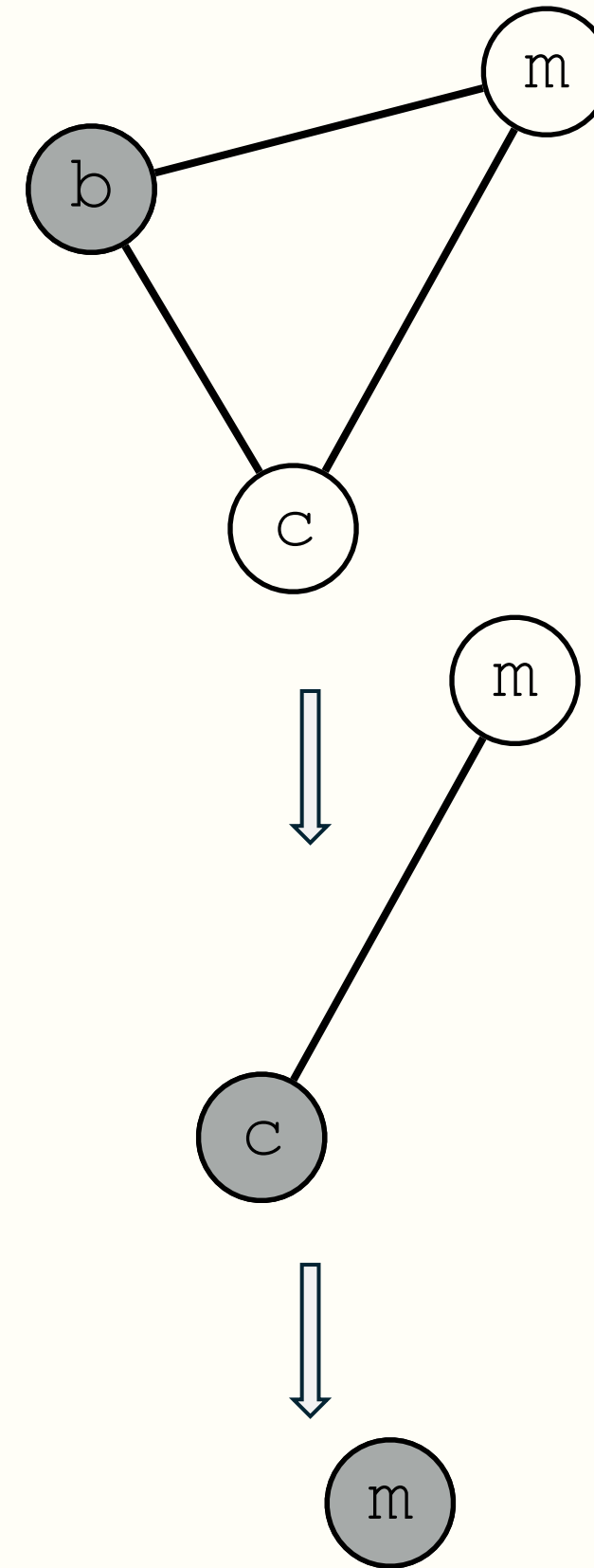
e

f

b

c

m

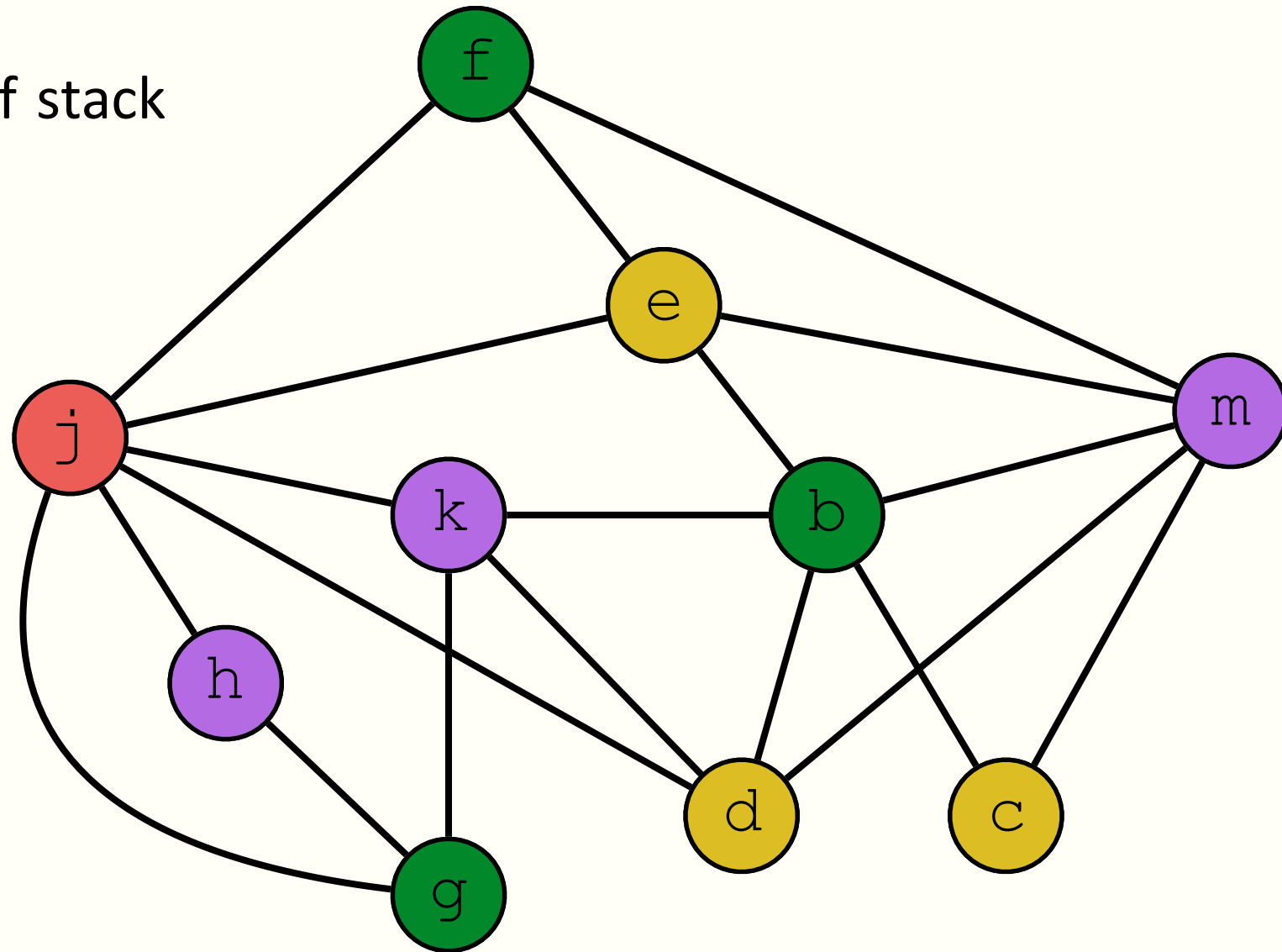


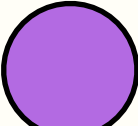
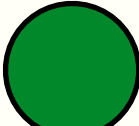
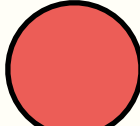
Simplification (4 registers)

Graph is now empty!
Color nodes in order of stack

Stack:

g
h
k
d
j
e
f
b
c
m



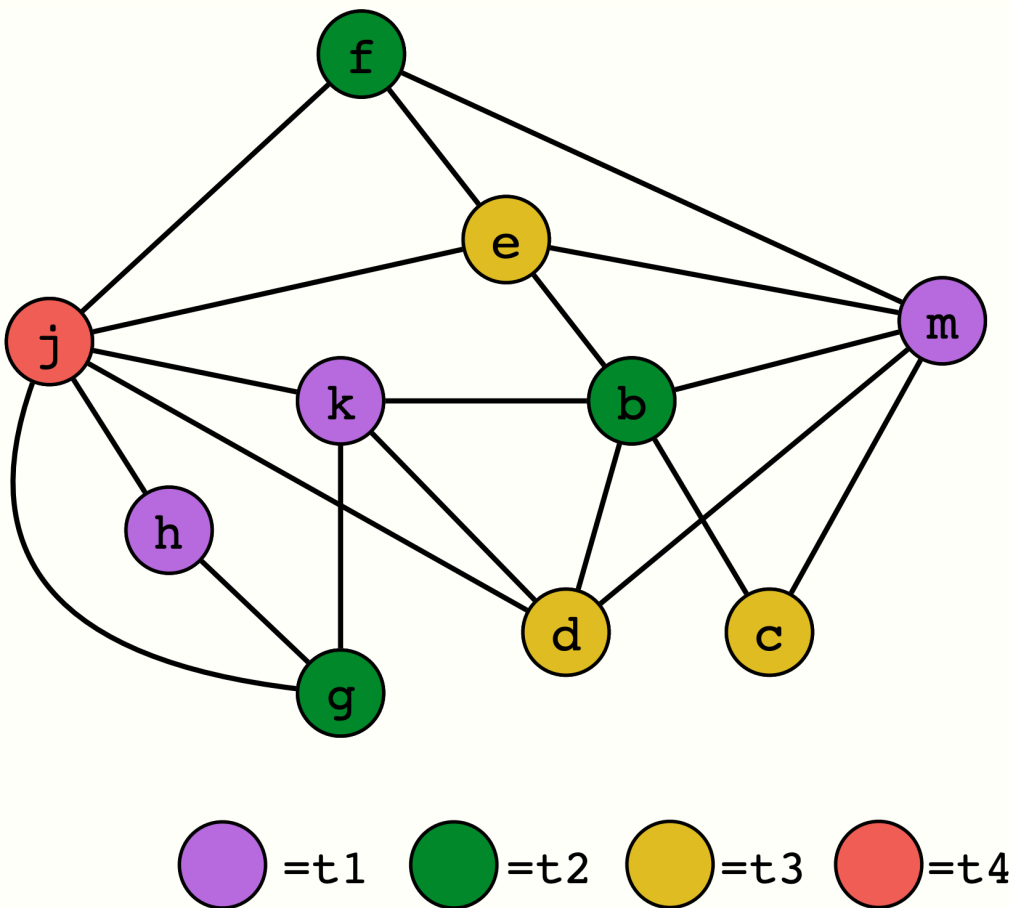
 =t1  =t2  =t3  =t4

Multiple valid colorings are possible!

Simplification (4 registers)

□ Transform the code based on the coloring

g := * (j+12)	t2 := * (t4+12)
h := k - 1	t1 := t1 - 1
f := g * h	t2 := t2 * t1
e := * (j+8)	t3 := * (t4+8)
m := * (j+16)	t1 := * (t4+16)
b := * (f+0)	t2 := * (t2+0)
c := e + 8	t3 := t3 + 8
d := c	t3 := t3
k := m + 4	t1 := t1 + 4
j := b	t4 := t2



Some moves might subsequently be simplified...

Stack Spilling

- ❑ The previous example worked out nicely!
- ❑ Always had nodes with degree $< k$
- ❑ Let's try again, but now with only **3 registers**...

{live-in: j, k}

`g := *(j+12)`

`h := k - 1`

`f := g * h`

`e := *(j+8)`

`m := *(j+16)`

`b := *(f+0)`

`c := e + 8`

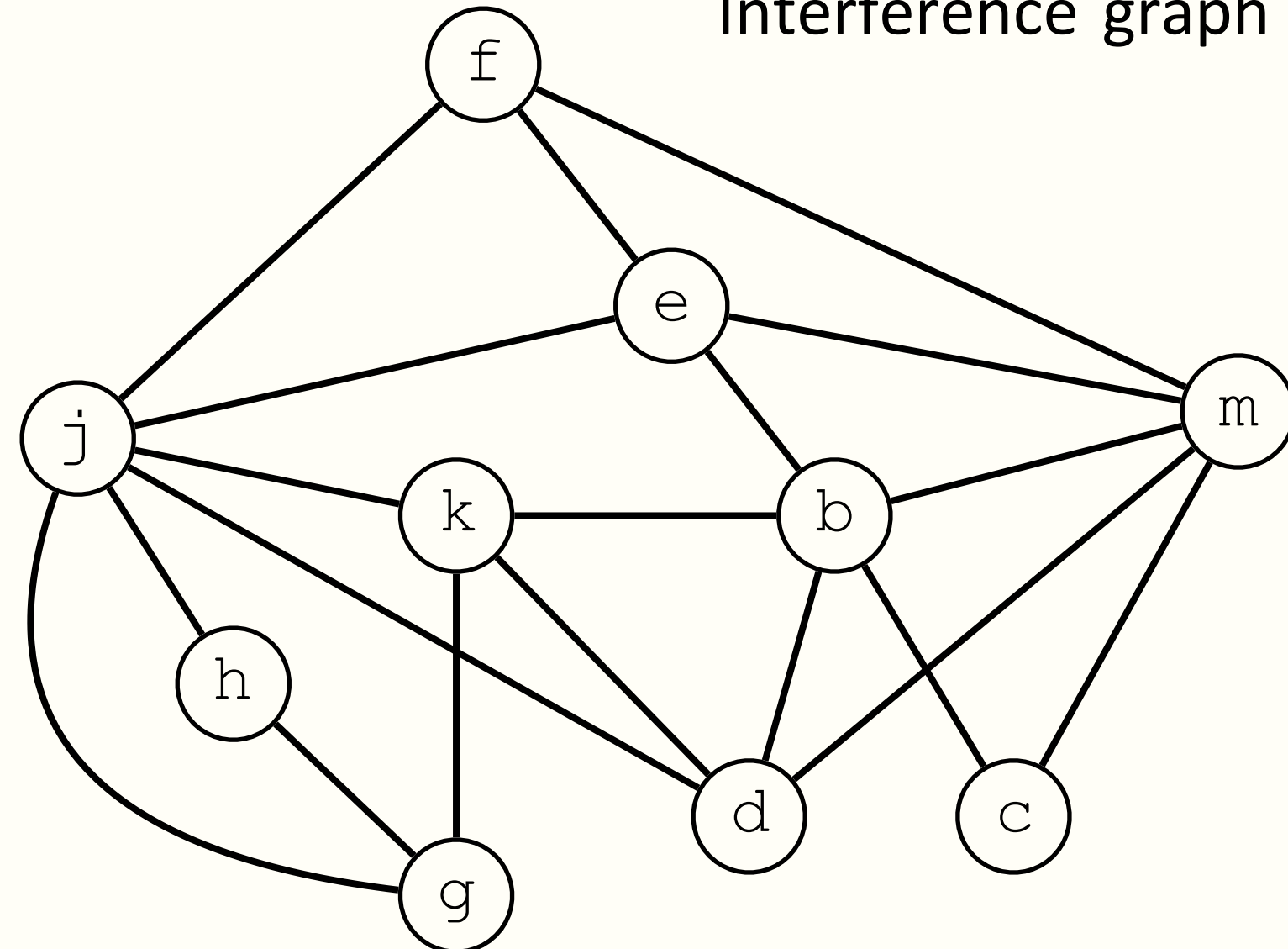
`d := c`

`k := m + 4`

`j := b`

{live-out: d, j, k}

Interference graph

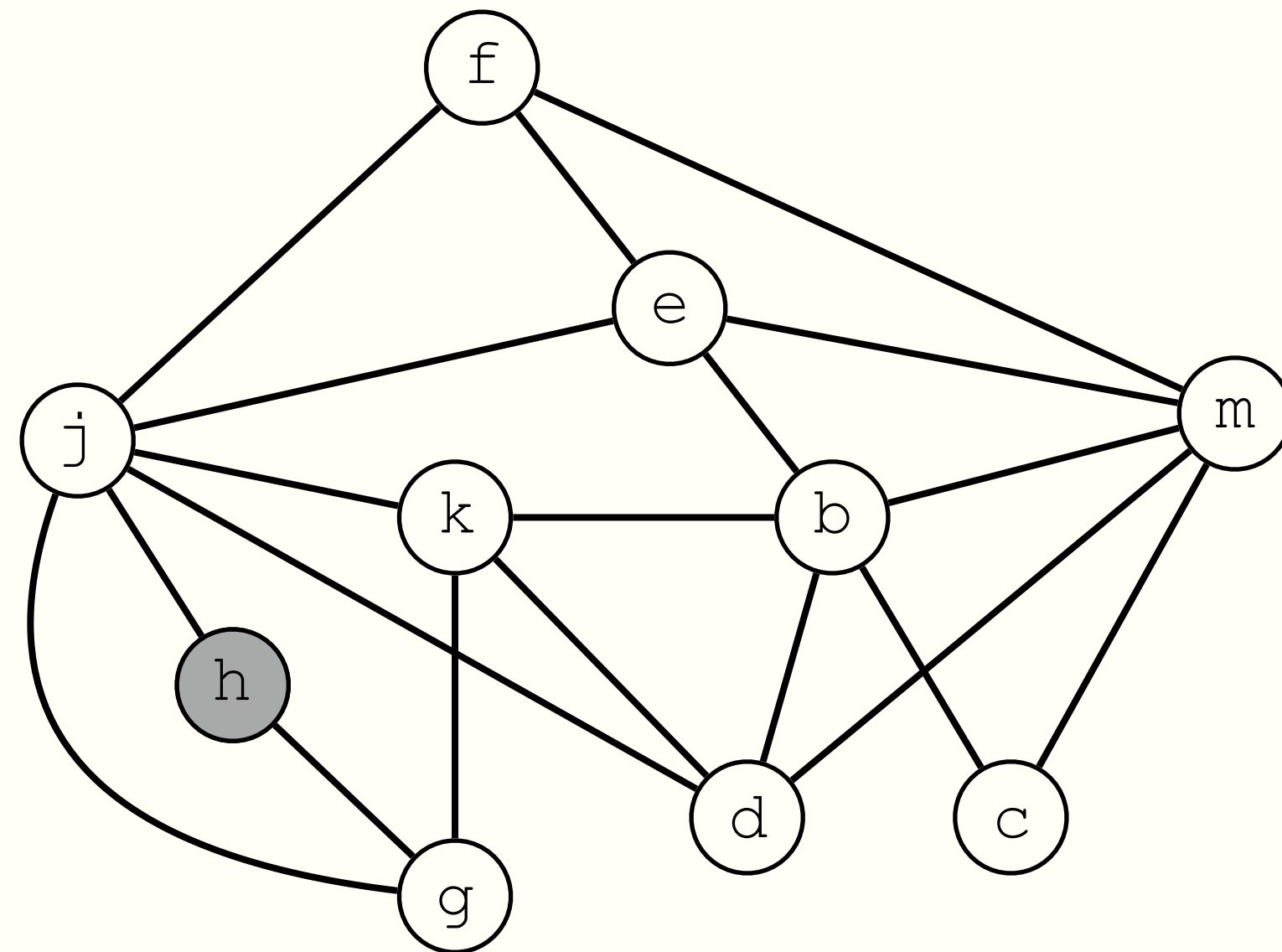


Simplification (3 registers)

Choose any node with degree < 3

Stack:

h



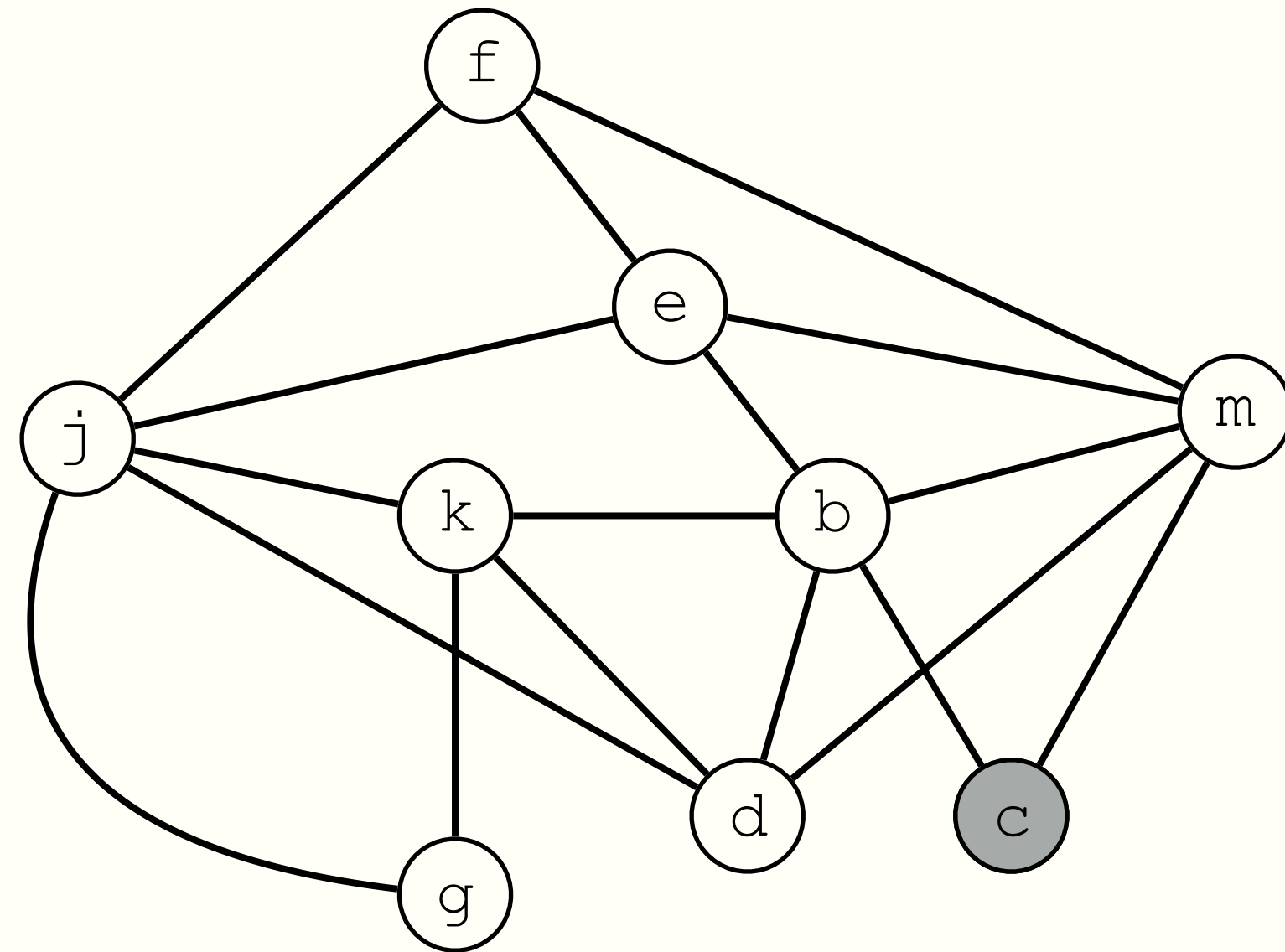
Simplification (3 registers)

Choose any node with degree < 3

Stack:

h

c



Simplification (3 registers)

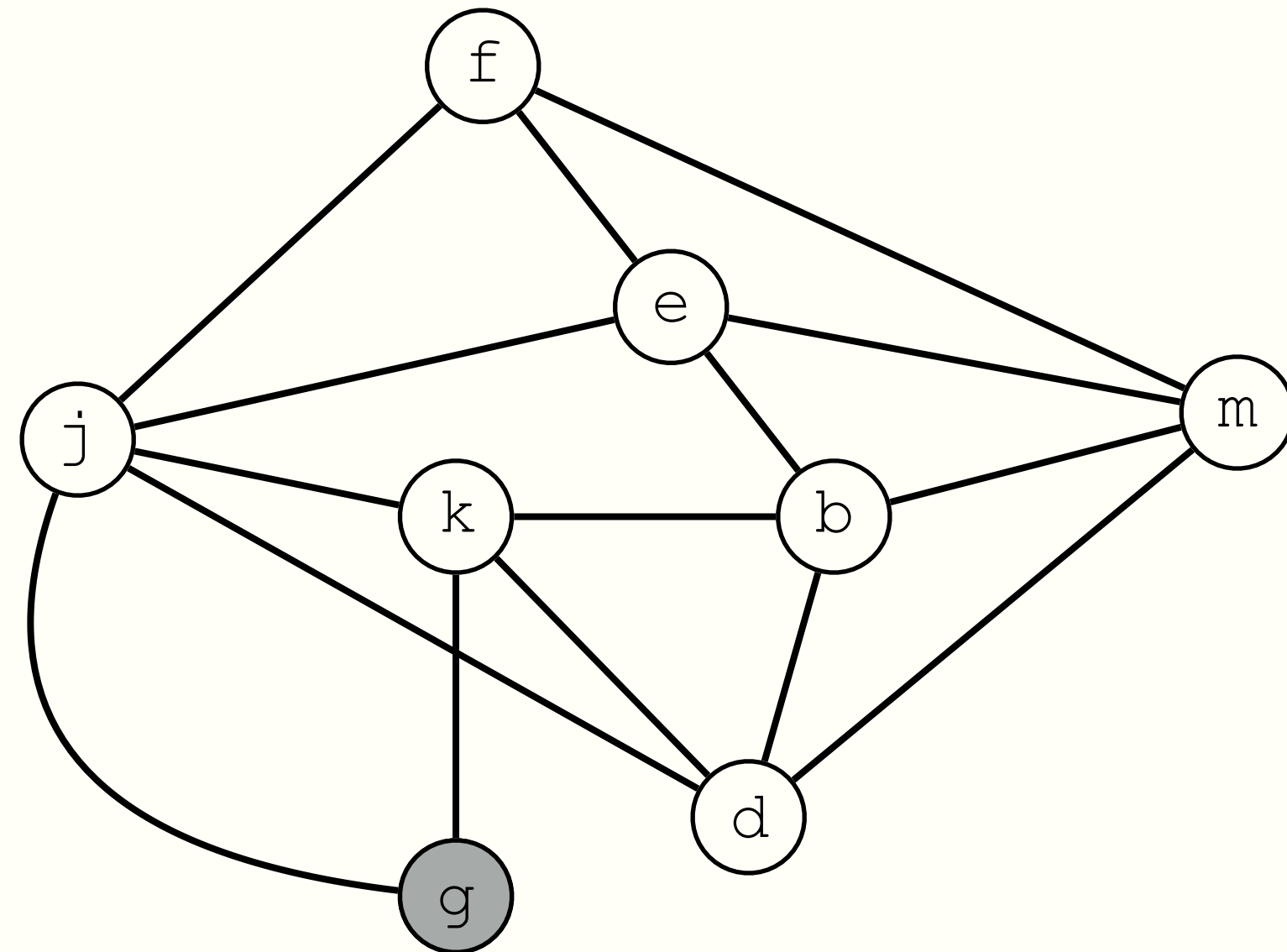
Choose any node with degree < 3

Stack:

h

c

g



Which Node to Spill?

❑ Now we are stuck! No nodes with degree <3

❑ Pick a node to **potentially spill**

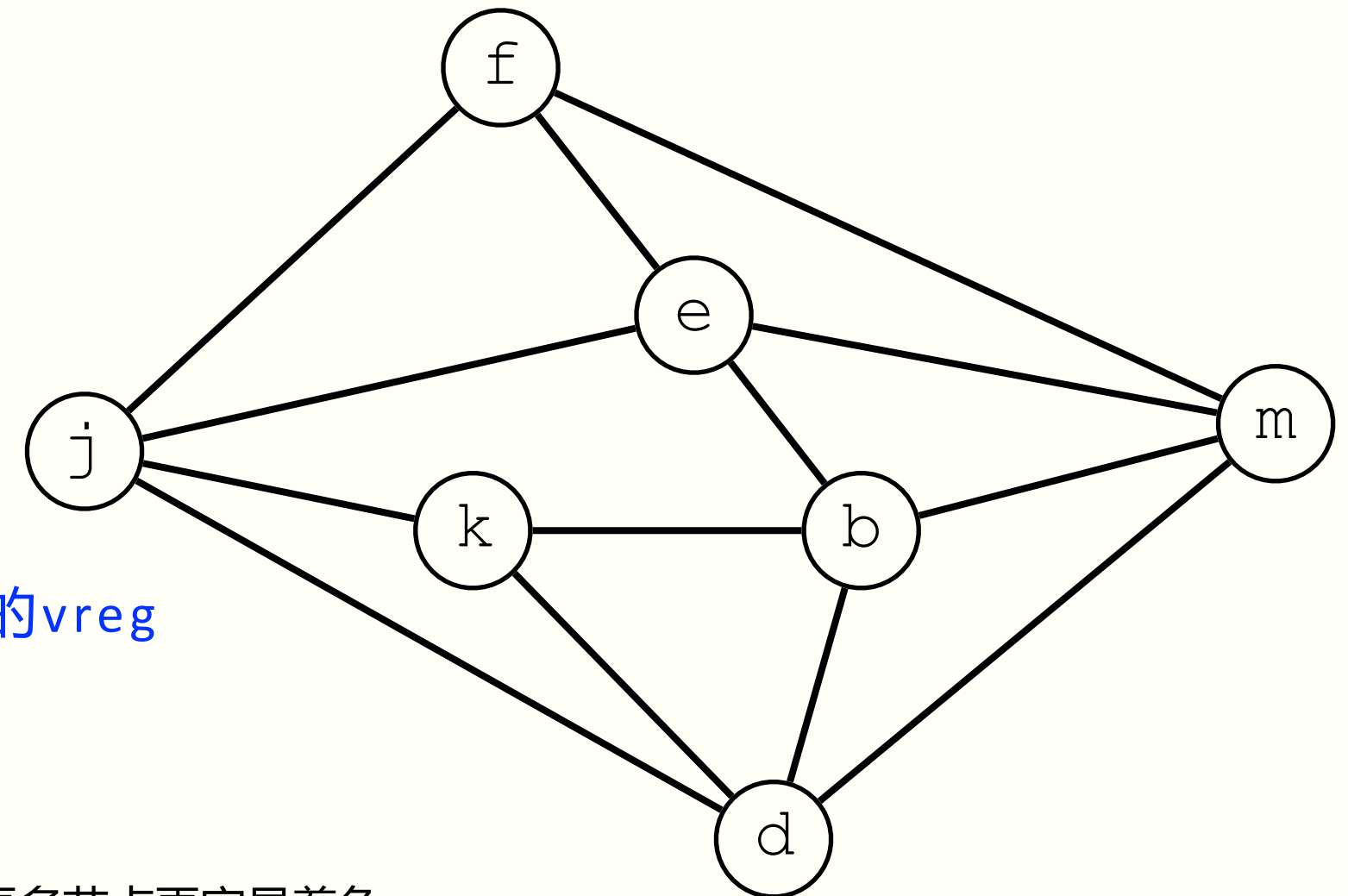
❑ Which node to spill?

- Pick a node with a **small priority value**
- 即spill代价较低、同时能够有效降低寄存器压力的vreg

❑ Compute **spill priority**

- Degree of node:
 - High degree: ($\approx$ live at many program points)
 - 度数大的节点与其他节点的寄存器冲突越多, spill它可以让更多节点更容易着色
- Not used/defined very often:
 - so don't need to access the stack very often

$$\text{Spill priority} = \frac{\text{Uses+defs outside loop} + \text{Uses+defs in loop} \times 10}{\text{degree of node}}$$



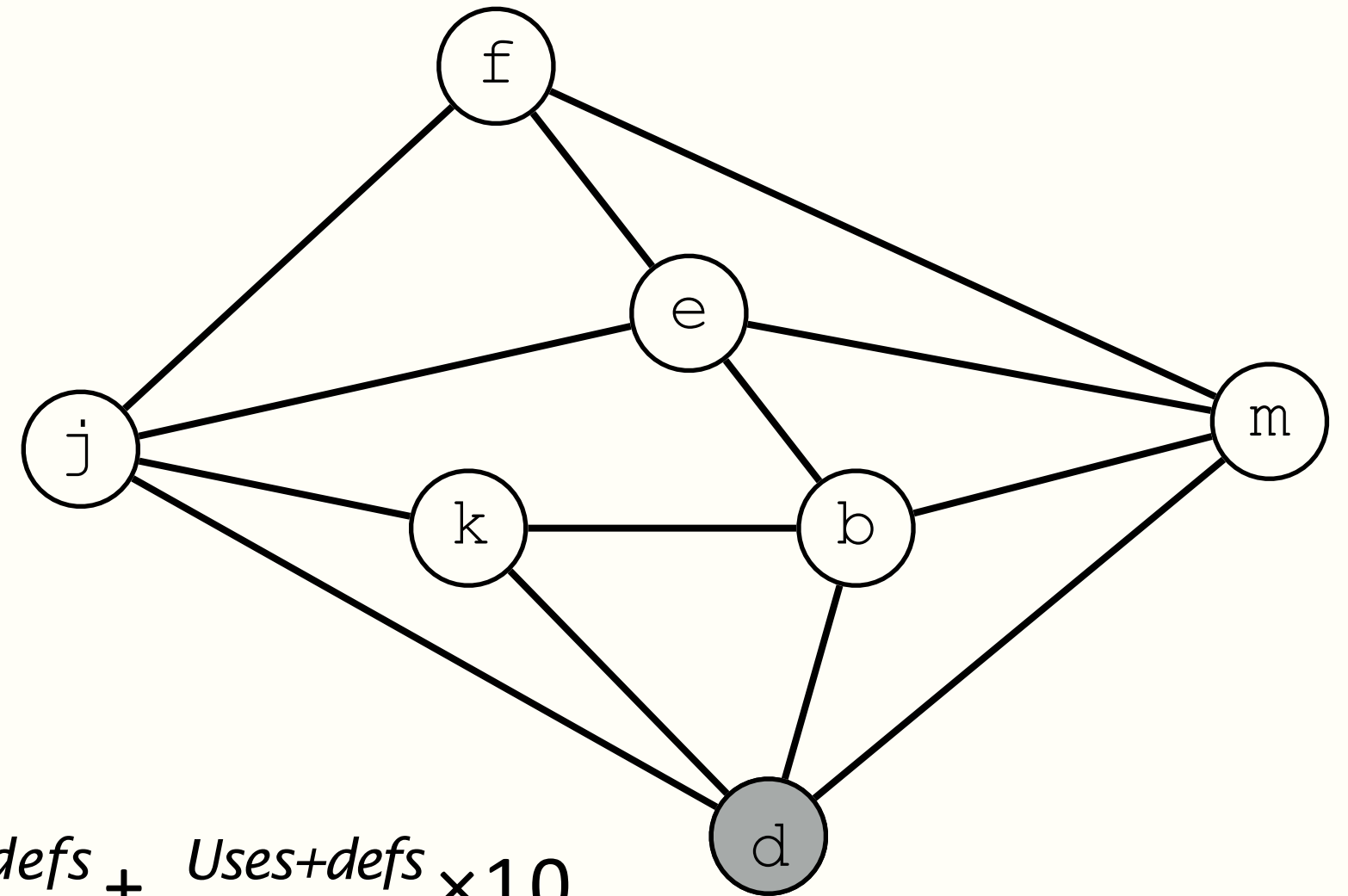
Used in our implementation.

$$\text{Spill priority} = \frac{\text{reference counts of Uses and Defs}}{\text{degree of node}}$$

Which Node to Spill?

```
{live-in: j, k}
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
{live-out: d, j, k}
```

Pick a node with a small spill priority
to potentially spill: **spill d?**



$$\text{Spill priority} = \frac{\text{Uses+defs}_{\text{outside loop}} + \text{Uses+defs}_{\text{in loop}} \times 10}{\text{degree of node}}$$

Simplification (3 registers)

Choose any node with degree < 3

Stack:

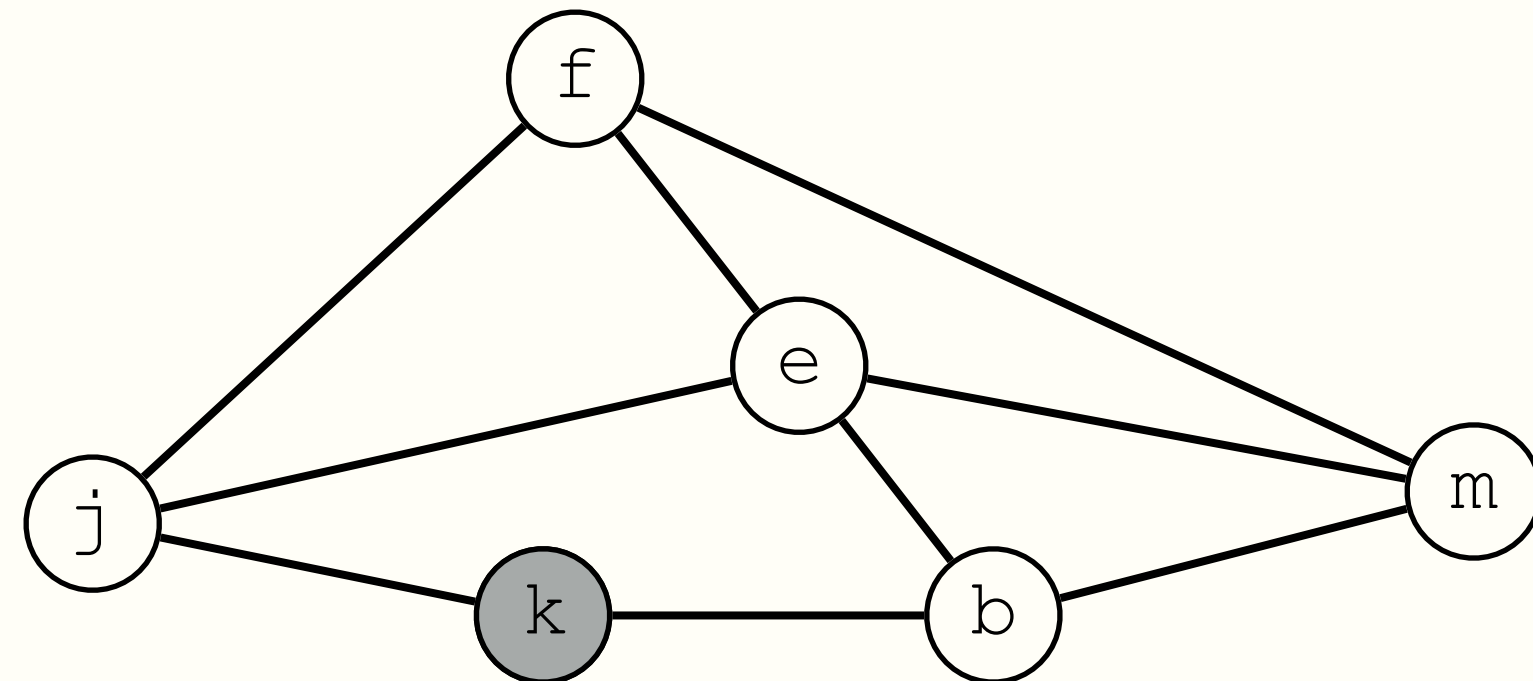
h

c

g

d *spill?*

k



Simplification (3 registers)

Choose any node with degree < 3

Stack:

h

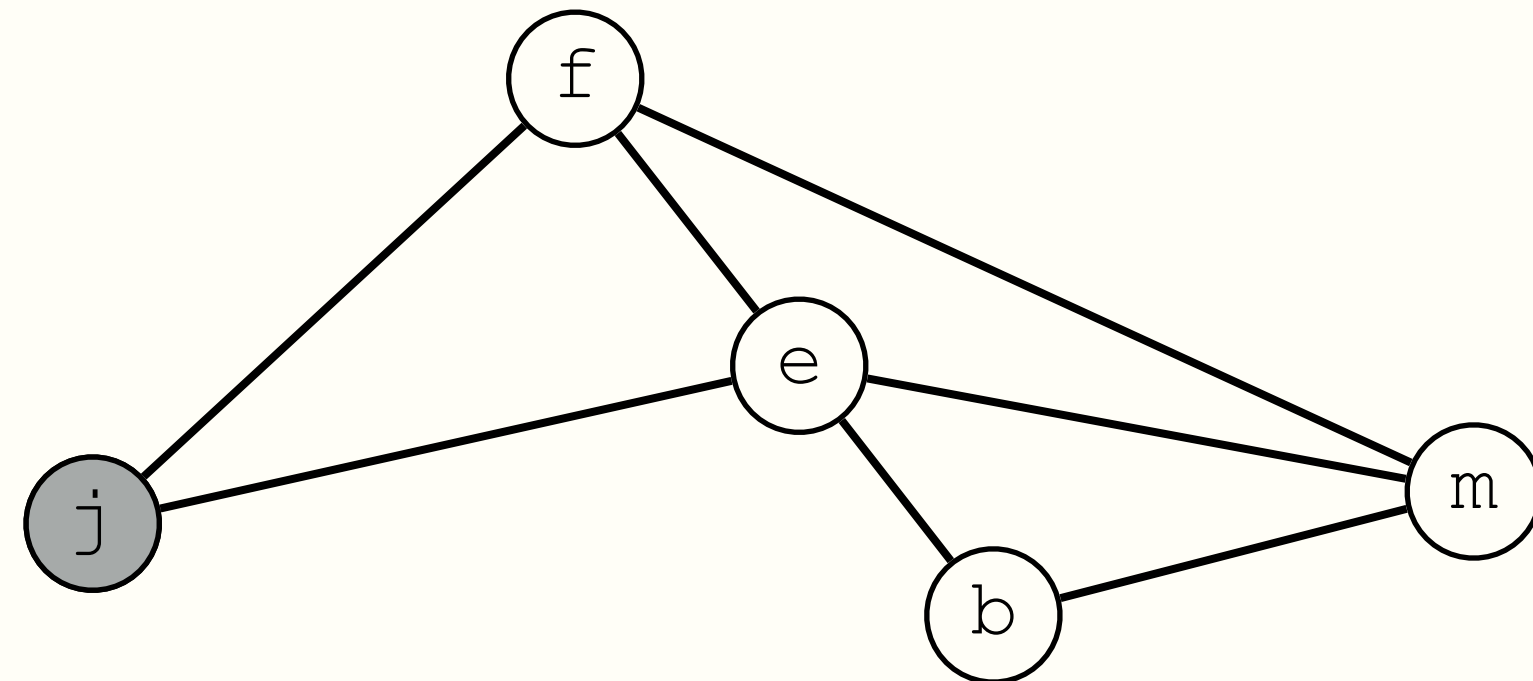
c

g

d *spill?*

k

j



Simplification (3 registers)

Choose any node with degree < 3

Stack:

h

c

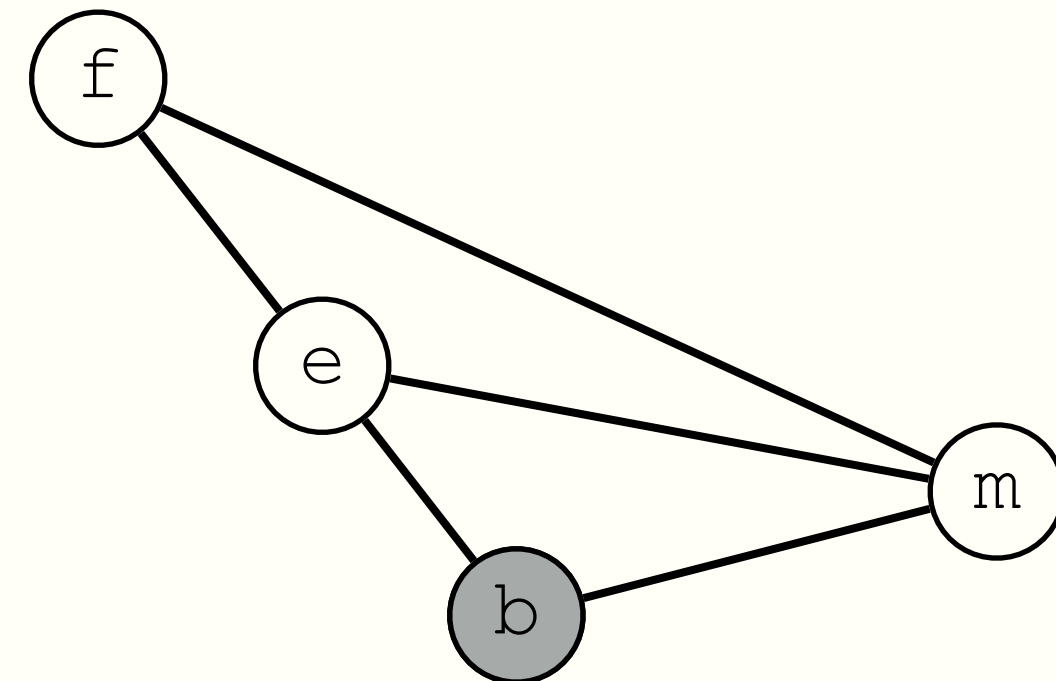
g

d *spill?*

k

j

b



Simplification (3 registers)

Choose any node with degree < 3

Stack:

h

c

g

d *spill?*

k

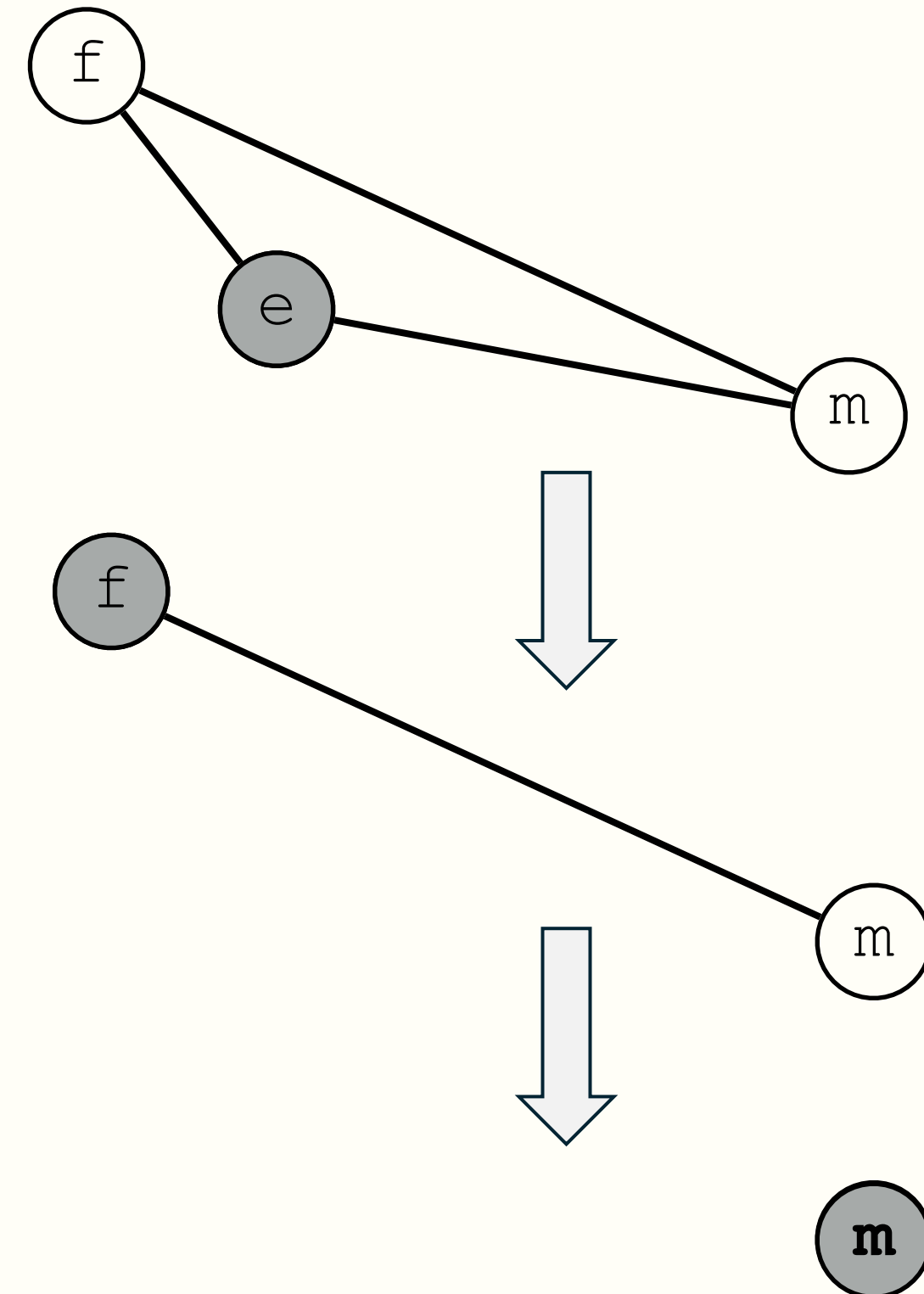
j

b

e

f

m



Select (3 registers)

Graph is now empty!
Color nodes in order of stack

Stack:

h

c

g

d *spill?*

k

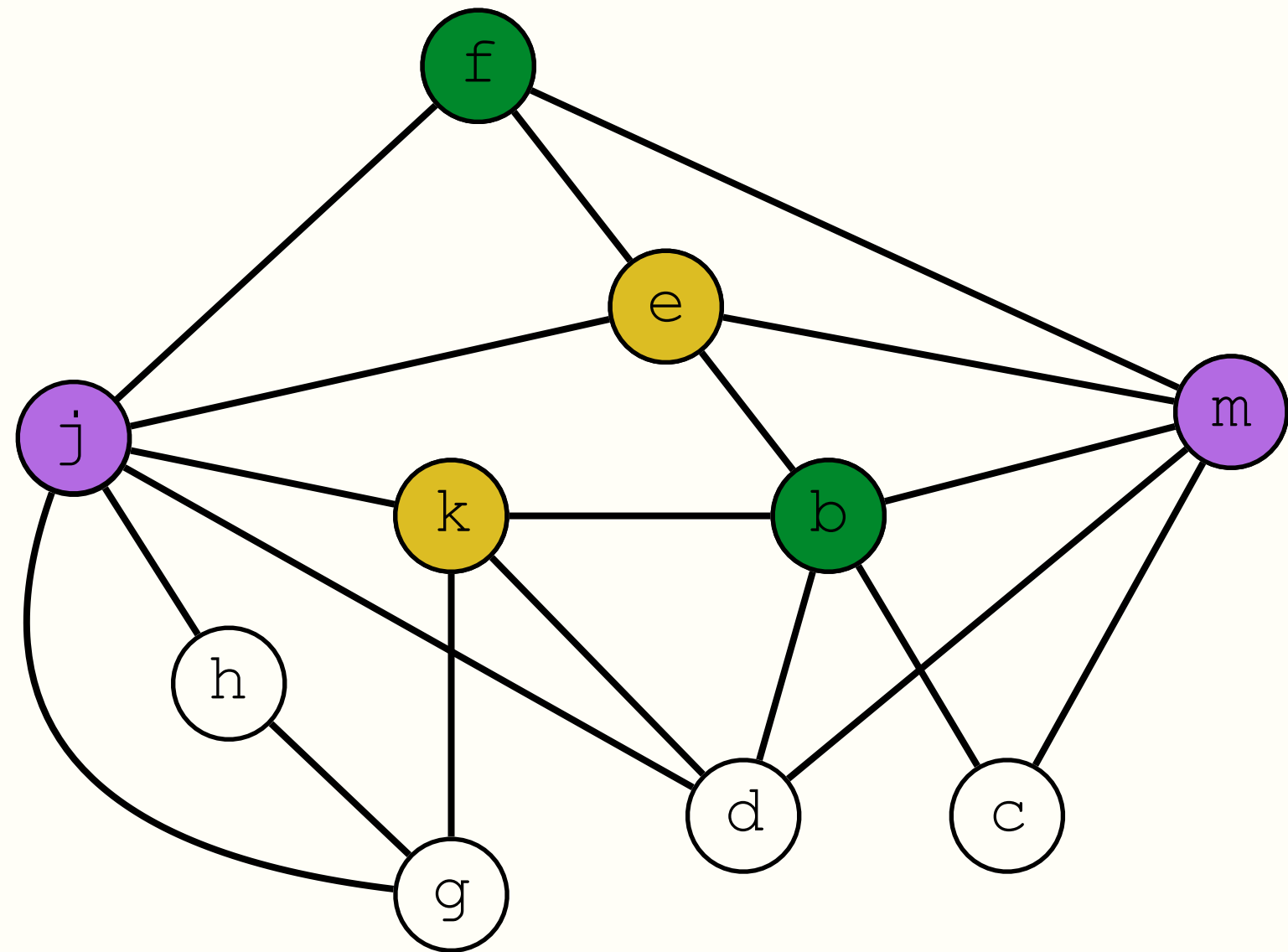
j

b

e

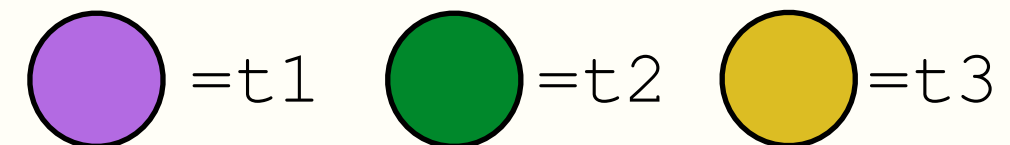
f

m



We got unlucky!

In some cases, a potential spill node is still colorable, and the Select phase can continue.



But in this case, we need to **rewrite**...

Select (3 registers)

□ Spill **d**

```
{live-in: j, k}
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
{live-out: d, j, k}
```



```
{live-in: j, k}
g := *(j+12)
h := k -
f := g *
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
*<fp+doff>:=d
k := m + 4
j := b
d2:=*<fp+doff>
{live-out: d2, j, k}
```

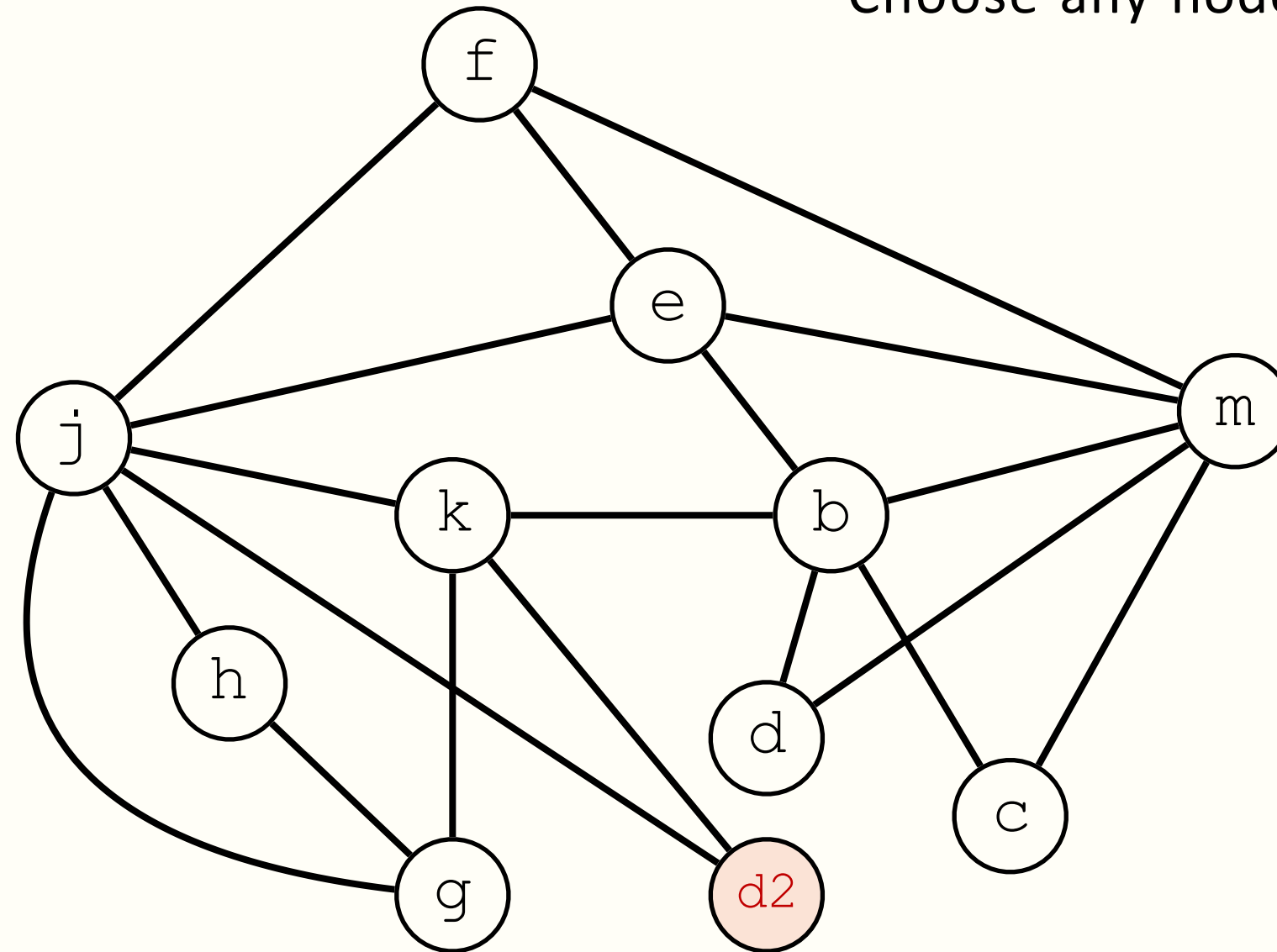
Build and Simplification (3 registers)

{live-in: j, k}

```
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
*<fp+doff>:=d
k := m + 4
j := b
d2:=*<fp+doff>
```

{live-out: d2, j, k}

Choose any node with degree <3



New Interference Graph

This time we succeed and will be able to complete the Select phase successfully!

Stack:

h
c
g
d
d2
k
b
m
e
f
j

Register Pressure

❑ Optimizations may increase register pressure:

- Copy propagation, common subexpression elimination (CSE), and loop-invariant code motion (LICM) can extend live ranges
- Longer live ranges may increase the number of simultaneously live virtual registers, potentially causing spills

❑ An alternative to copy propagation:

- Let the register allocator assign the same physical register to copy-related virtual registers when safe.
- Redundant copy instructions can then be eliminated.
- This can avoid unnecessarily extending live ranges.

❑ Takeaway:

- Reducing instruction count does not always improve performance; register pressure must also be considered

Coalescing Register Allocation

□ If we have “ $x := y$ ” and x and y have no edge in the interference graph, we might be able to assign them the same color.

● This would translate to “ $r_i := r_i$ ”, which would then be removed

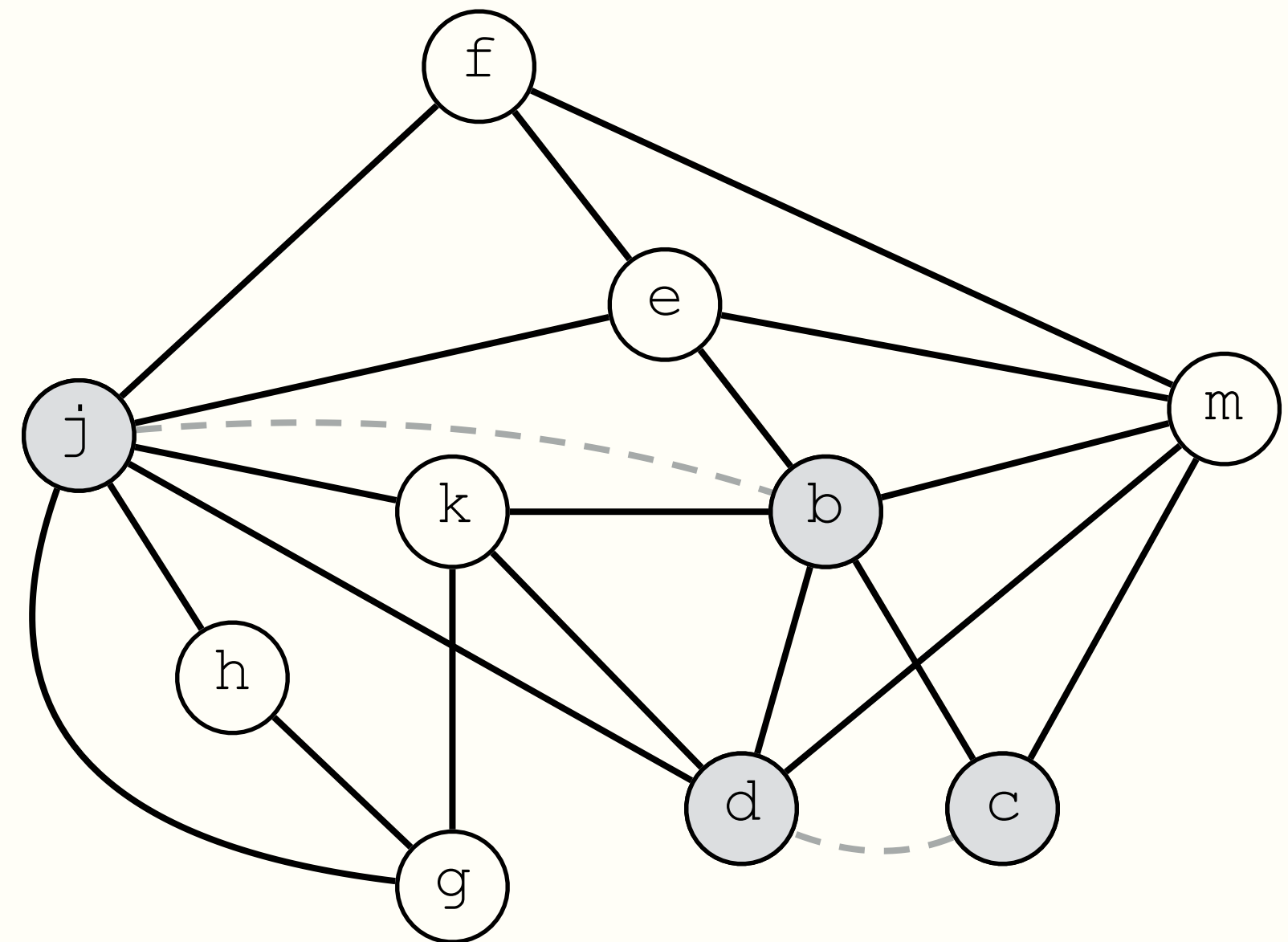
□ Idea: optimistically coalesce nodes in the interference graph

● Just take the edges to be the union

● E.g., following nodes could be coalesced

● d and c, j and b

```
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
```



Coalescing Heuristics

- ❑ But coalescing may make a k -colorable graph **uncolorable!**
- ❑ **Briggs:** safe to coalesce x and y if the resulting node n will have fewer than k neighbors with degree $\geq k$.
 - 节点 n 的所有低度数邻居 (degree $< k$) 均可被 Simplify
 - 此时 n 的剩余邻居数 $< k$, 因此 n 也可被 Simplify
 - 剩余图是原图的子图, 因此原图若 k -colorable, 剩余图也必然 k -colorable
- ❑ **George:** safe to coalesce x and y if for every neighbor t of x , either t already interferes with y or t has degree $< k$
 - 若邻居 t 已与 y 干涉, 合并不会增加其颜色约束
 - 否则, 若 degree(t) $< k$, 合并后 t 度数不变, 仍可安全着色
 - 合并不会破坏原图的 k -colorability
- ❑ These strategies are conservative
 - Will not turn a k -colorable graph into a non- k -colorable graph

Coloring with Coalescing

❑ **Build:** construct an interference graph

- Categorize nodes as *move-related* (if src or dest of a move) or *non-move-related*

❑ **Simplify:** Remove non-move-related nodes with degree $< k$

❑ **Coalesce:** Coalesce nodes using Briggs' or George's heuristic

- Possibly re-mark coalesced nodes as *non-move-related*
- Continue with Simplify if there are nodes with degree $< k$

❑ **Freeze:** if there is a low-degree ($< k$) move-related node, freeze it

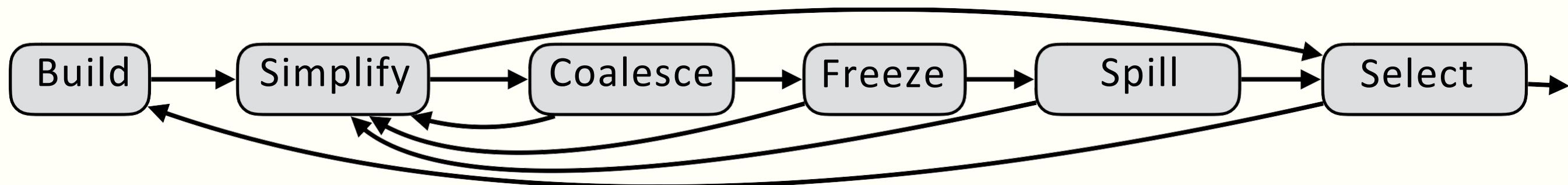
- i.e., make it non-move-related, i.e., give up on coalescing that node
- Continue with Simplify

❑ **Spill:** choose node with degree $\geq k$ to potentially spill

- Then continue with simplify

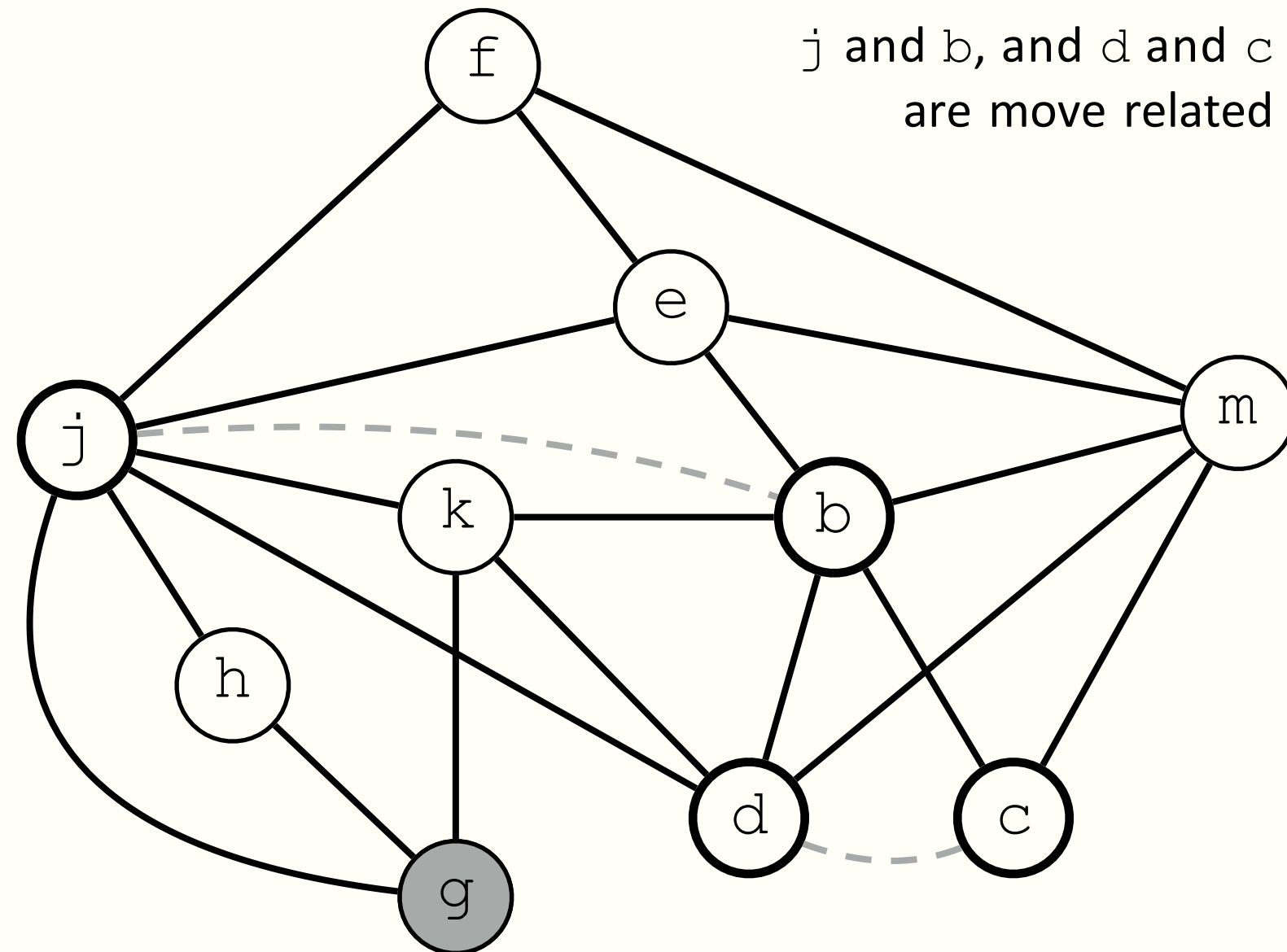
❑ **Select:** when graph is empty, start restoring nodes in reverse order and color them

- Potential spill node: try coloring it; if that fails, rewrite the program to use a stack and try again!



Example (4 registers)

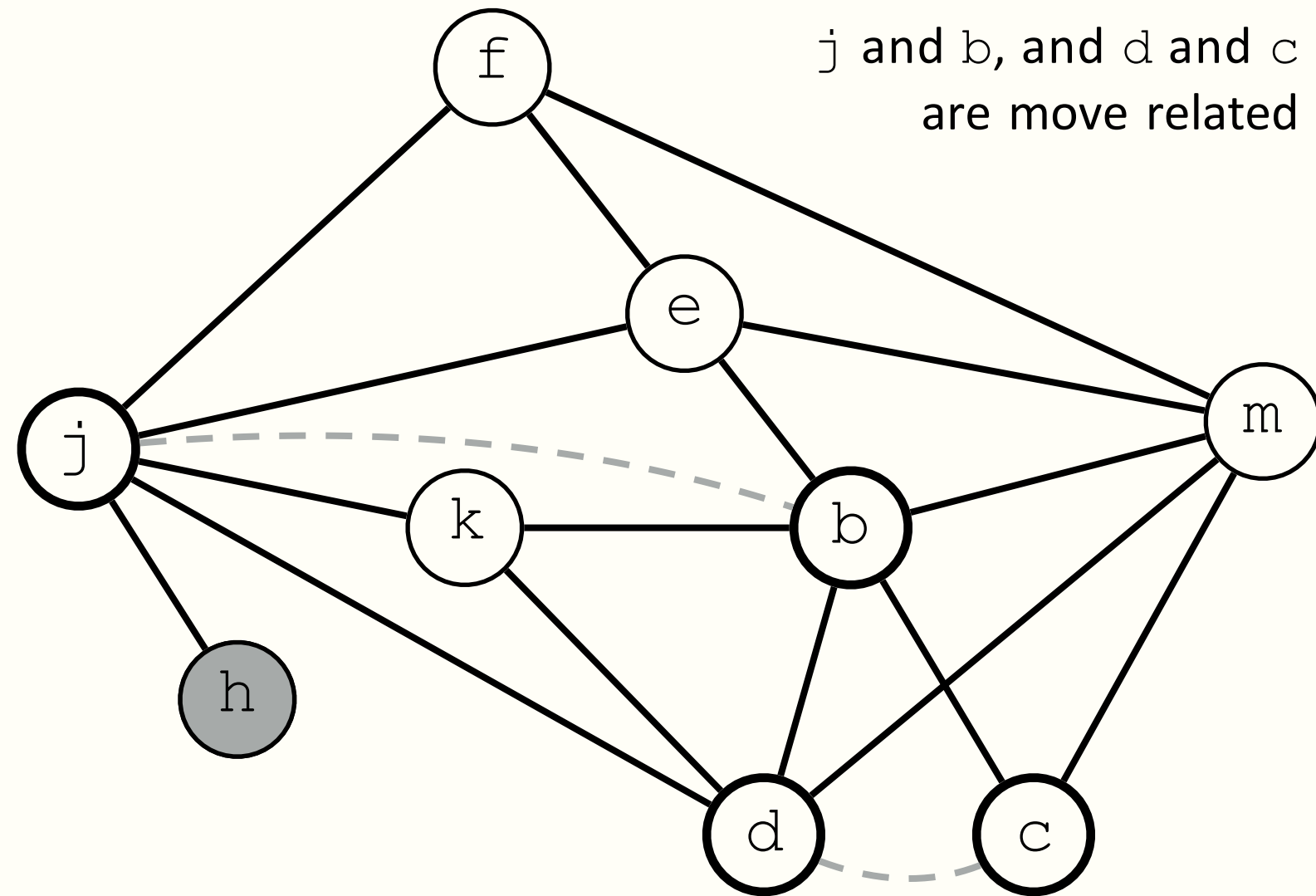
Stack:
g



Example (4 registers)

Stack:

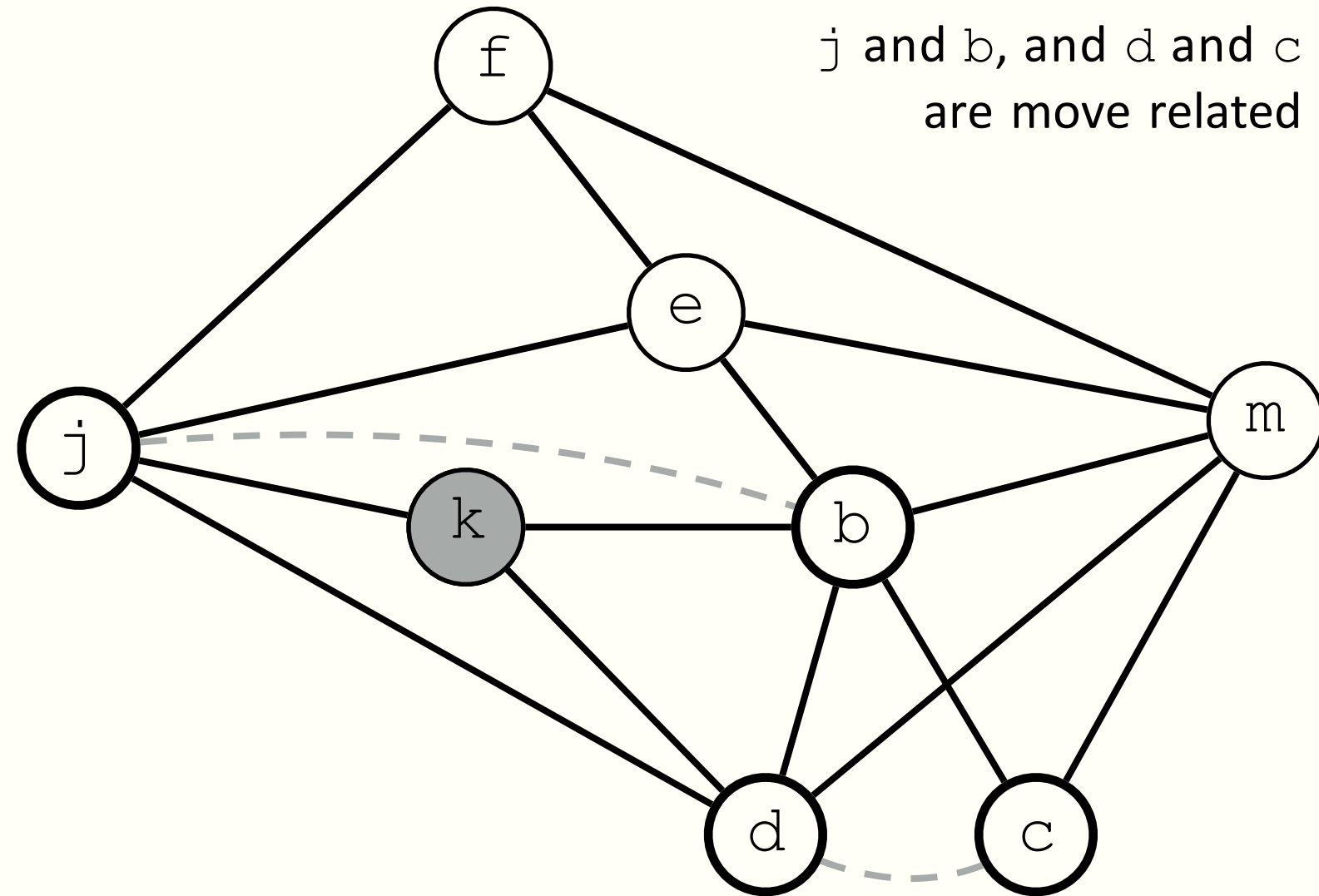
g
h



Example (4 registers)

Stack:

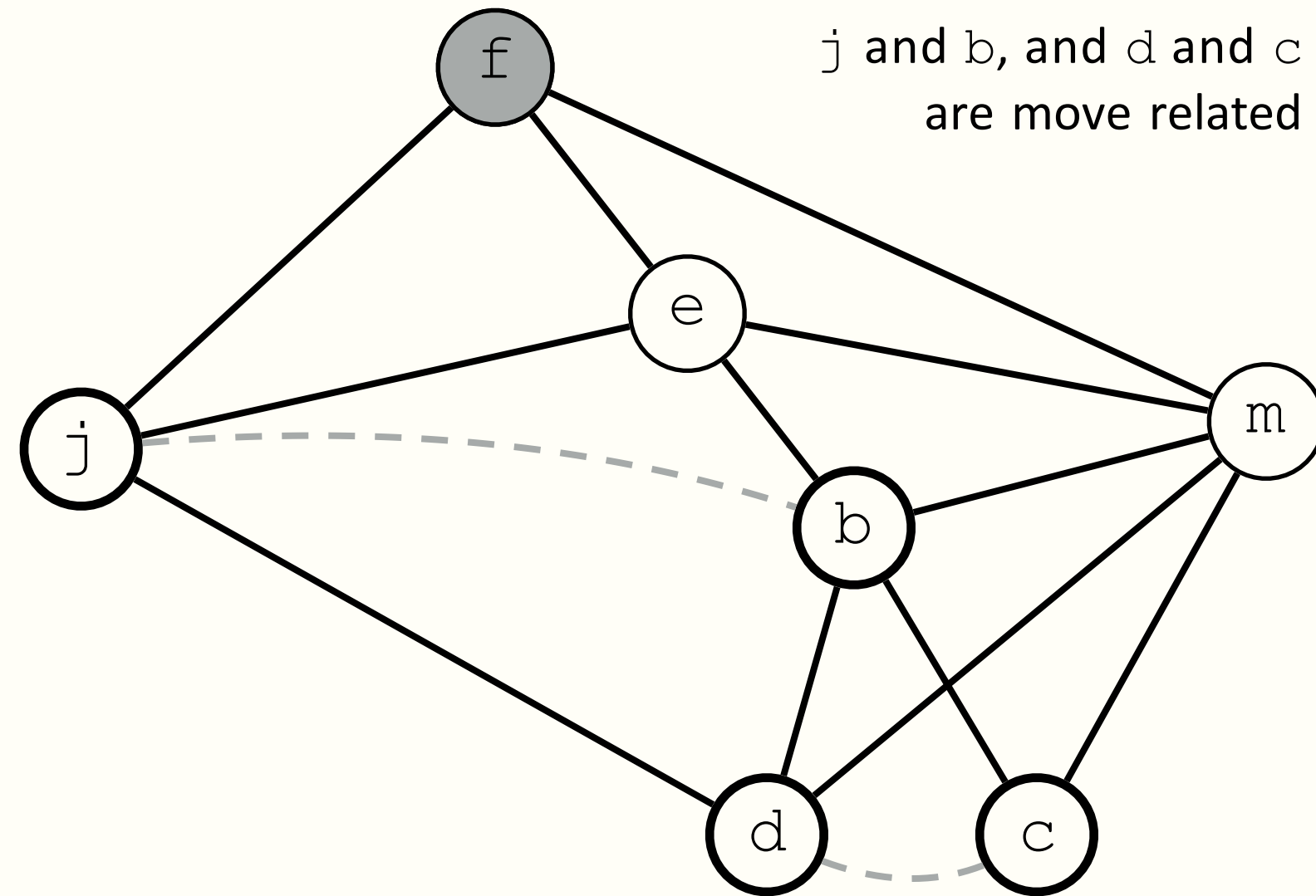
g
h
k



Example (4 registers)

Stack:

g
h
k
f

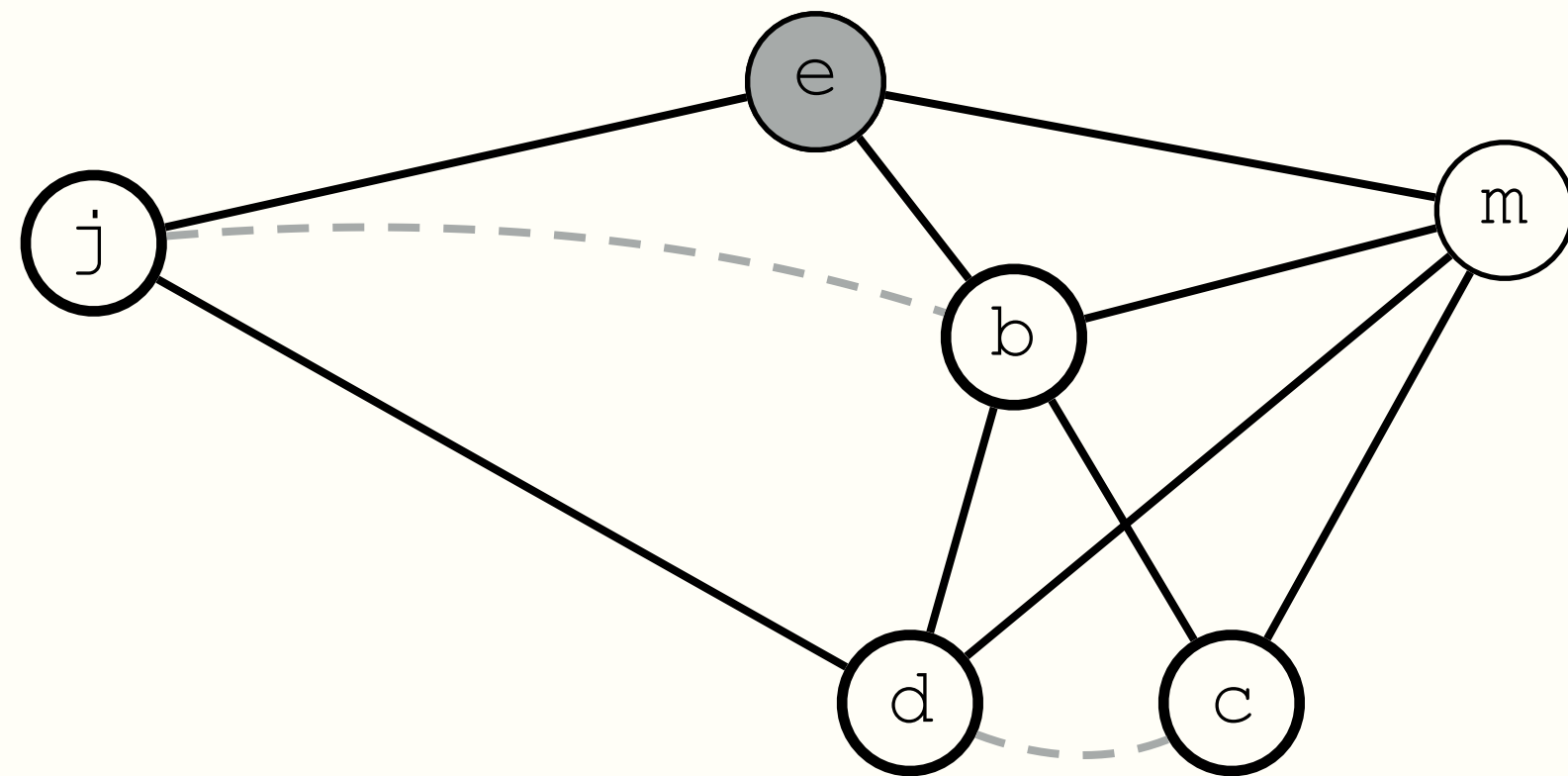


Example (4 registers)

Stack:

g
h
k
f
e

j and b, and d and c
are move related

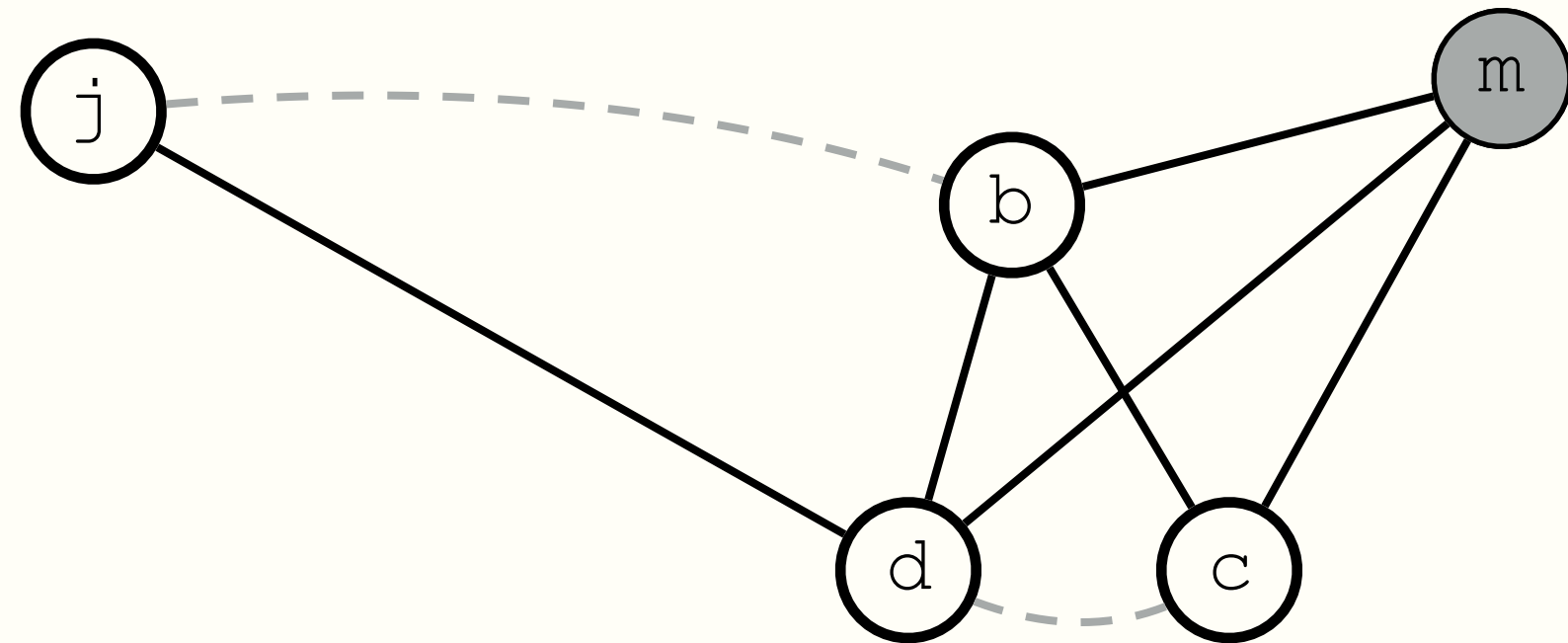


Example (4 registers)

Stack:

g
h
k
f
e
m

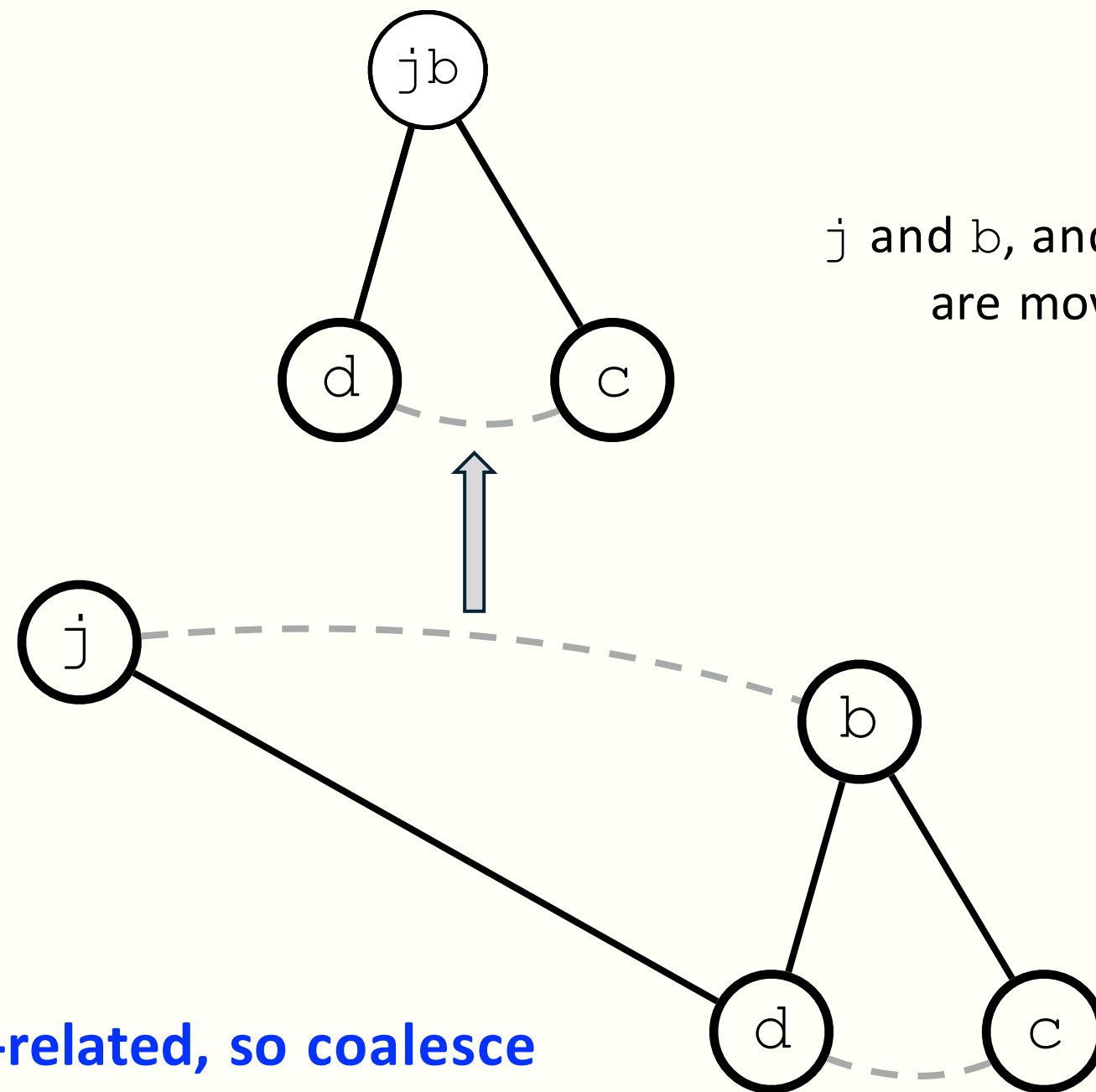
j and b, and d and c
are move related



Example (4 registers)

Stack:

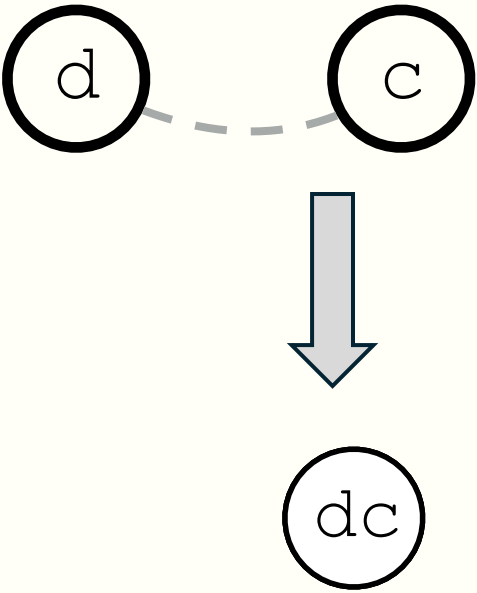
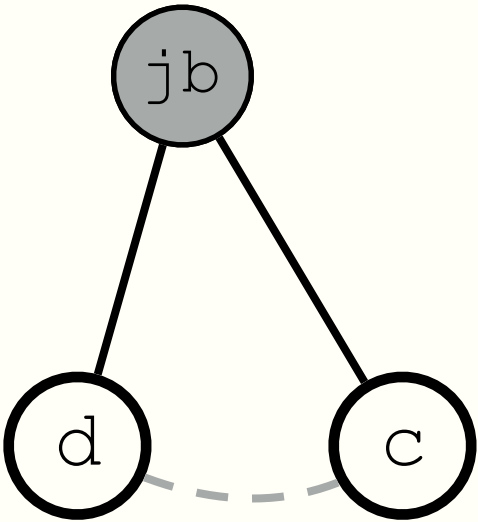
g
h
k
f
e
m



Example (4 registers)

Stack:

g
h
k
f
e
m
jb
dc



j and b, and d and c
are move related

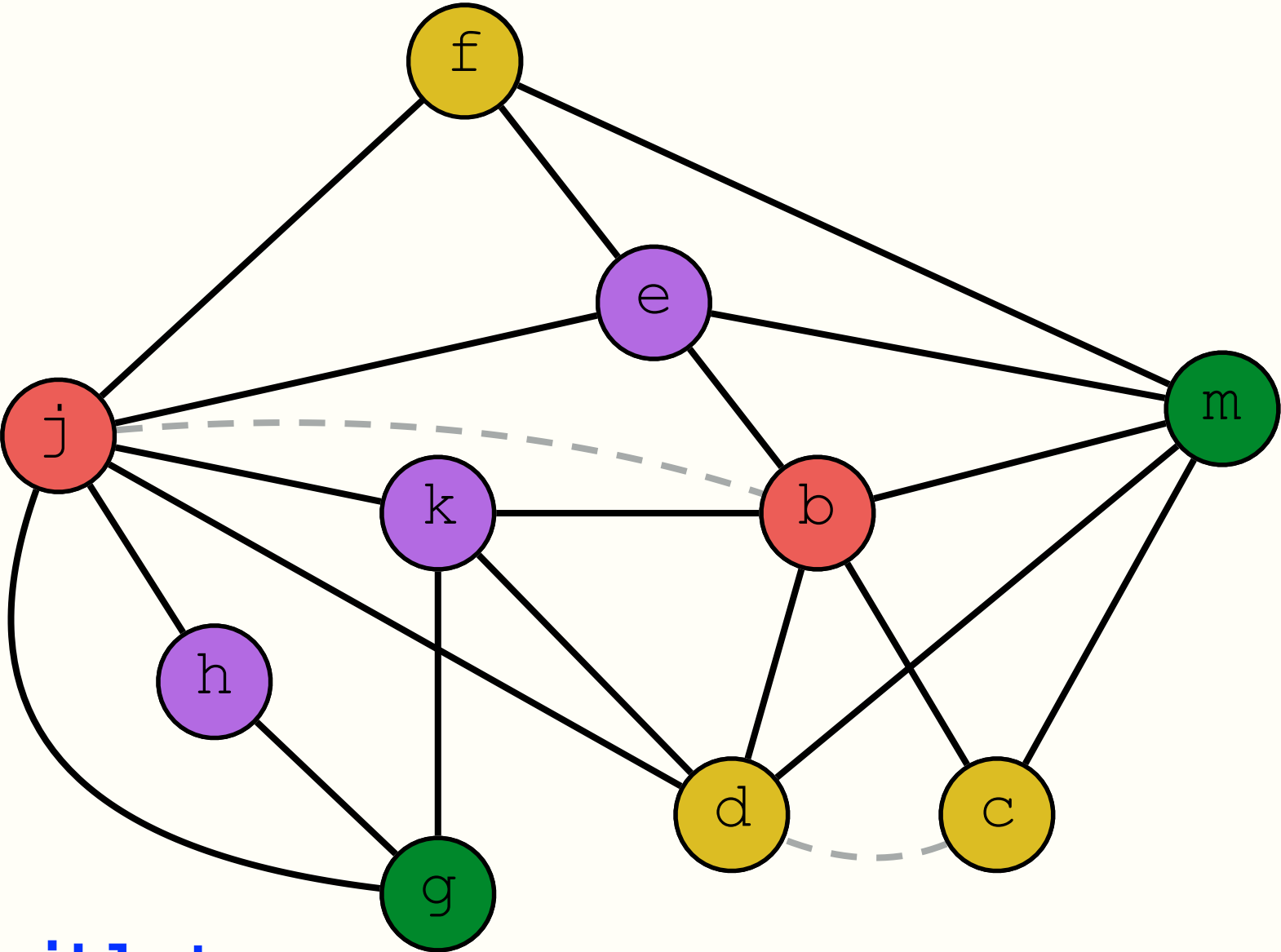
Example (4 registers)

Stack:

g
h
k
f
e
m
jb
dc



=t1 =t2 =t3 =t4



Multiple valid colorings are possible!

Example (4 registers)

● = t1 ● = t2 ● = t3 ● = t4

{live-in: j, k}

g := * (j+12)

h := k - 1

f := g * h

e := * (j+8)

m := * (j+16)

b := * (f+0)

c := e + 8

d := c

k := m + 4

j := b

{live-out: d, j, k}



{live-in: t4, t1}

t2 := * (t4+12)

t1 := t1 - 1

t3 := t2 * t1

t1 := * (t4+8)

t2 := * (t4+16)

t4 := * (f+0)

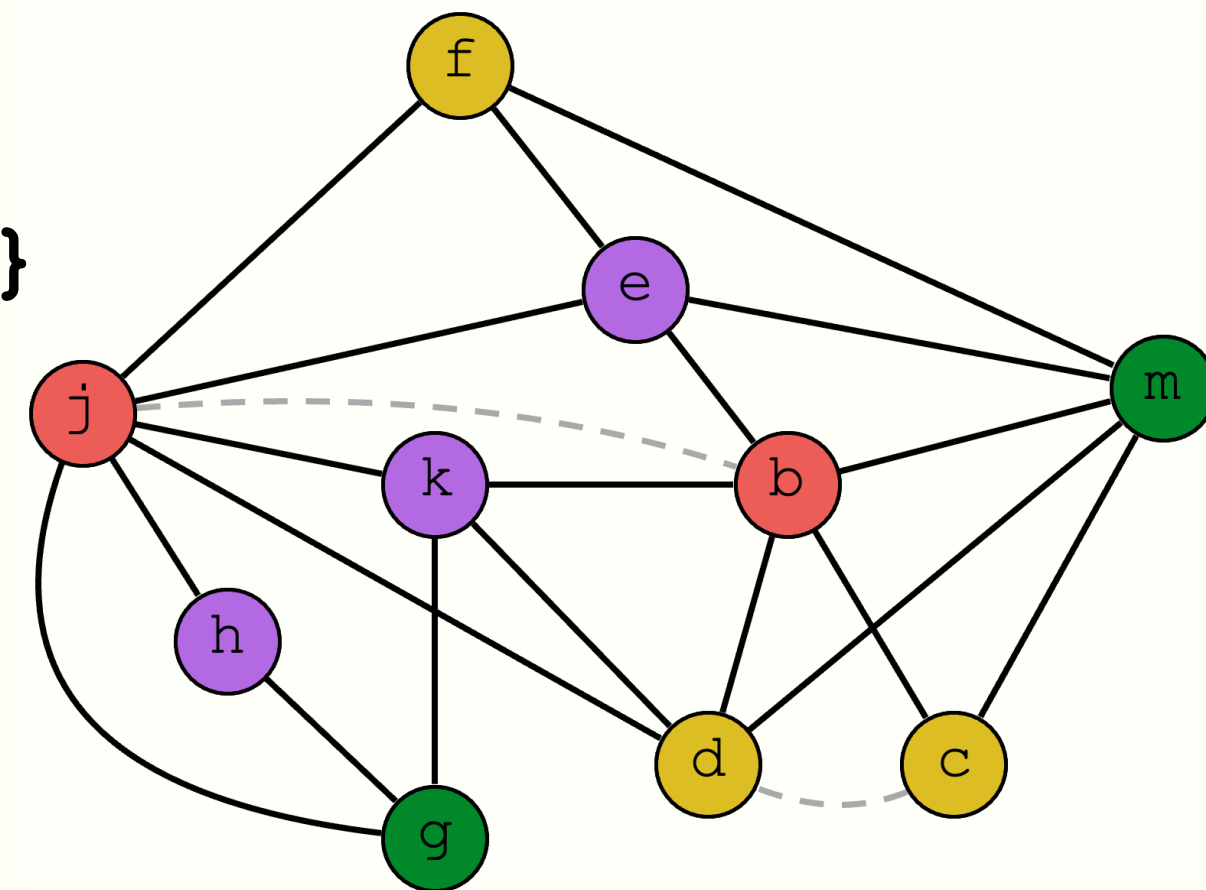
t3 := t1 + 8

t3 := t3

t1 := t2 + 4

t4 := t4

{live-out: t3, t4, t1}



This is the result
of coalescing!

Pre-colored Registers

- ❑ The MIR may explicitly use physical registers
 - e.g., RISC-V registers a0, a1, ra, sp, etc
 - This allows us to model calling-convention constraints
- ❑ Physical registers are treated as pre-colored nodes
 - Each pre-colored node has a fixed register assignment
 - Allocatable pre-colored registers may be shared with non-interfering virtual registers
 - (Pre-colored 不代表寄存器被永久独占, 只代表这个节点的颜色已经固定)
- ❑ We cannot simplify a pre-colored node:
 - Simplification assumes a node can be assigned any available color
 - Pre-colored nodes have fixed colors, so they cannot be freely re-colored.
- ❑ Similarly, we cannot spill a pre-colored node
 - Their assignments to physical registers must be preserved.

Using Physical Registers (RISC-V)

❑ At Procedure Boundaries

- Move arguments from a0–a7 (or stack slots) into fresh virtual registers
- Move the return value into a0 (or a0–a1 when required)

❑ Within the Procedure Body

- Use virtual registers instead of physical registers
- Minimize the live ranges of pre-colored values

❑ During Register Allocation

- Coalescing may eliminate unnecessary move instructions
- Reuse incoming argument stack slots for spills when safe

```
int f(int x, int y) {  
    return x + y;  
}
```



```
f:  
mv v1, a0  
mv v2, a1  
add v3, v1, v2  
mv a0, v3
```



```
f:  
add a0, a0, a1  
ret
```



```
f:  
mv a0, a0  
mv a1, a1  
add a0, a0, a1  
mv a0, a0  
ret
```

Before Register Allocation

Register Allocation & Coalescing

Redundant Move Elimination

Calling Conventions in Register Allocation (RISC-V)

❑ Register Palette: {a0–a7, t2–t6, s1–s11}, k = 24

- **Reserved:** zero, ra, sp, gp, tp (also s0/fp, t0–t1 in our implementation)
- **Caller-saved** (a0–a7, t2–t6): may be overwritten by a function call
- **Callee-saved** (s1–s11): must be preserved if modified

❑ Handling Caller-saved Registers

- Treat a function call as defining all caller-saved registers
- Add interference edges between these registers and virtual registers live across the call
- This prevents live-across-call values from being assigned to caller-saved registers

❑ Handling Callee-saved Registers

- Callee-saved registers can be assigned to virtual registers
- Save modified callee-saved registers on entry and restore them before returning
- Model save/restore using fresh virtual registers, allowing coalescing or spilling

Graph Coloring 本身并不会自动产生 **Callee-saved** 的保存/恢复指令。编译器需要在代码生成时显式建模，或在寄存器分配后生成相应的 **Prologue/Epilogue** 代码。

Example

□ Compile the following C function

- Assume the target machine has 3 registers
- \$r1 and \$r2 are caller-save
- \$r3 is callee-save

```
int f(int a, int b) {  
    int d = 0;  
    int e = a;  
    do {  
        d = d+b;  
        e = e-1;  
    } while (e > 0);  
    return d;  
}
```

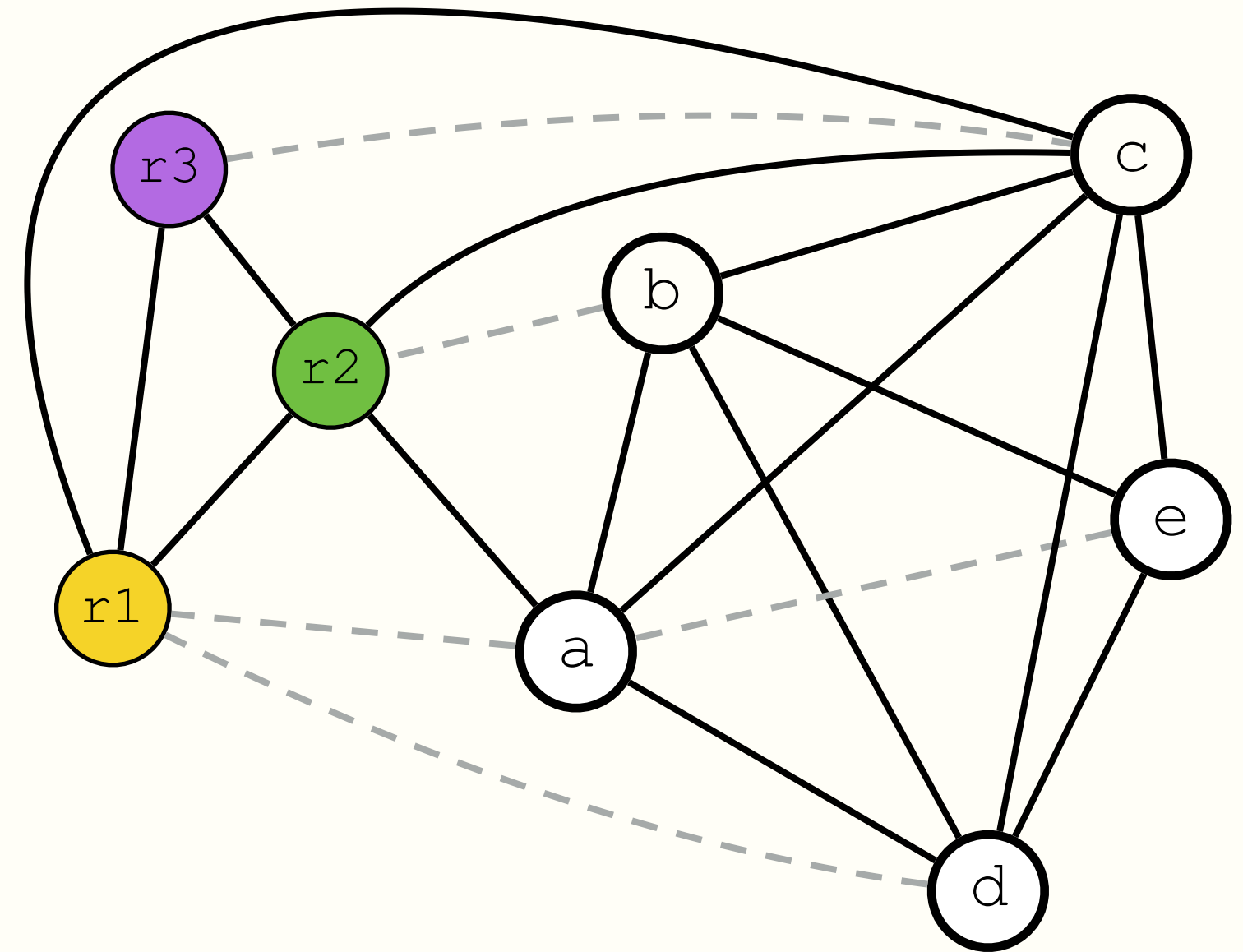
```
f: c := $r3    ; preserve callee  
    a := $r1    ; move arg1 into a  
    b := $r2    ; move arg2 into b  
    d := 0  
    e := a  
loop:  
    d := d + b  
    e := e - 1  
    if e > 0 loop else end  
end:  
    r1 := d      ; return d  
    r3 := c      ; restore callee  
    return      ; $r3,$r1 live out
```

Example

□ Build the inference graph

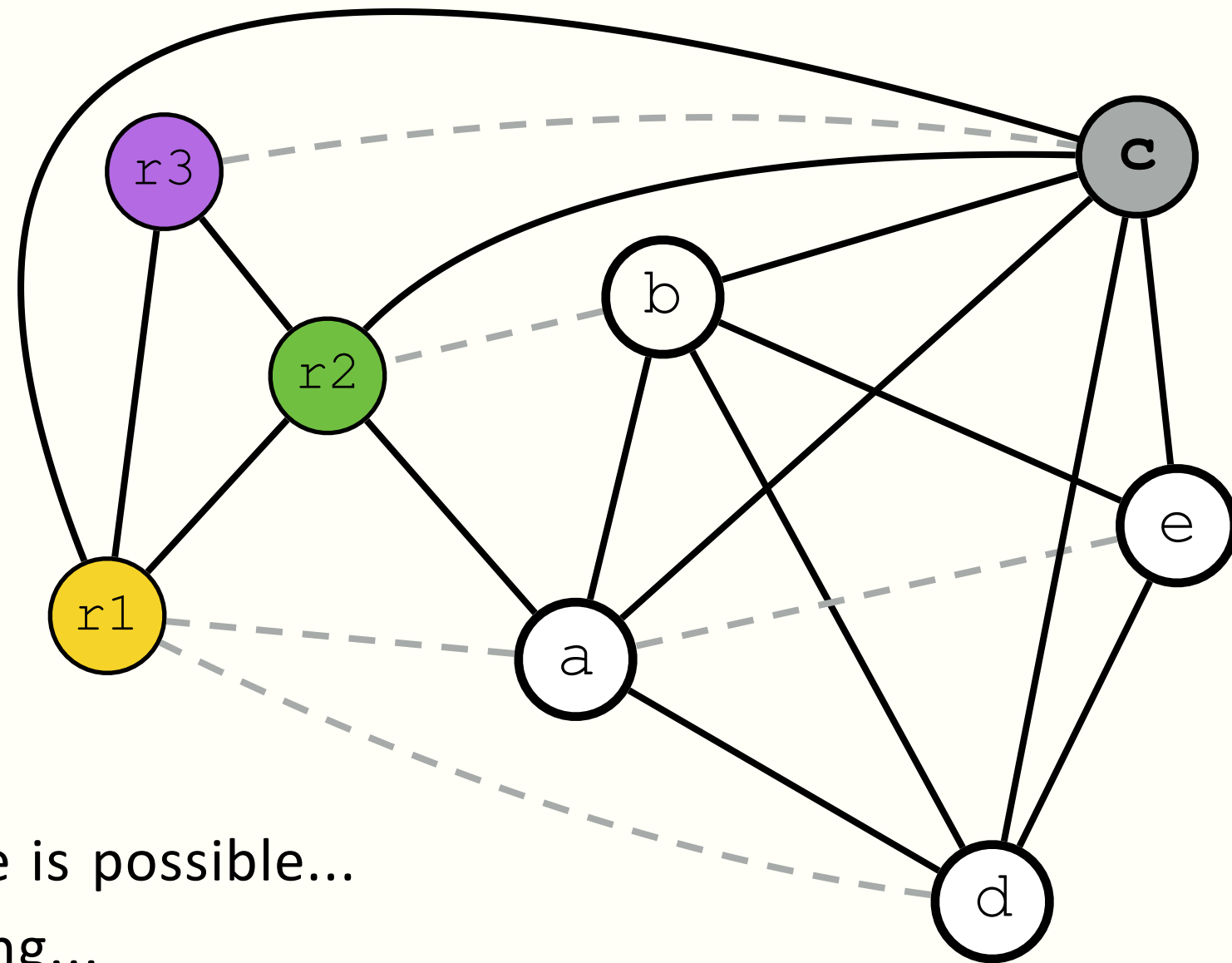
- r1, r2, and r3 are pre-colored

```
f: c := $r3 ; preserve callee
  a := $r1 ; move arg1 into a
  b := $r2 ; move arg2 into b
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 loop else end
end:
  r1 := d ; return d
  r3 := c ; restore callee
return ; $r3,$r1 live out
```



Example: 3 registers (r1, r2, r3)

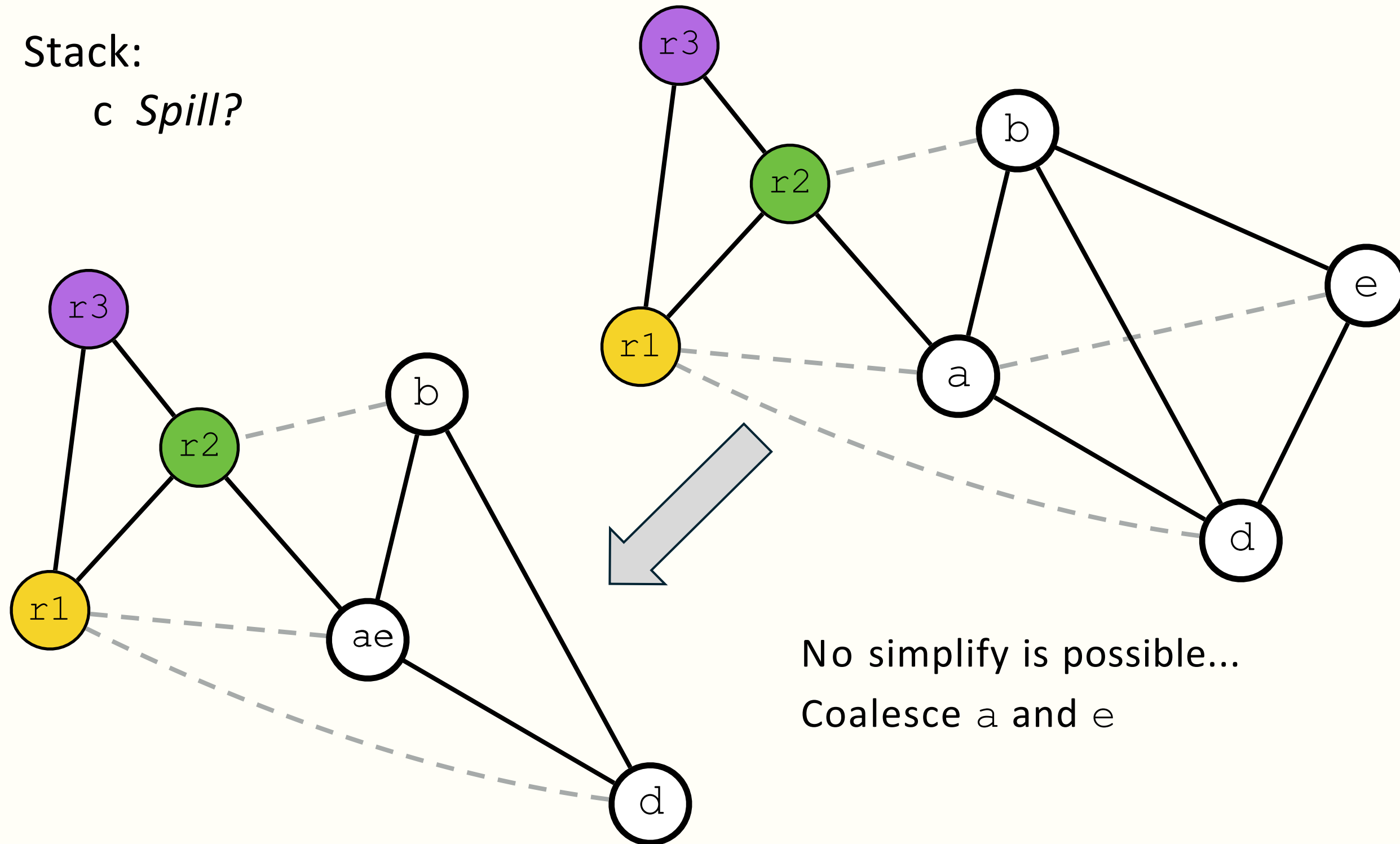
Stack:
c Spill?



No simplify, coalesce, or freeze is possible...
c is a good candidate for spilling...

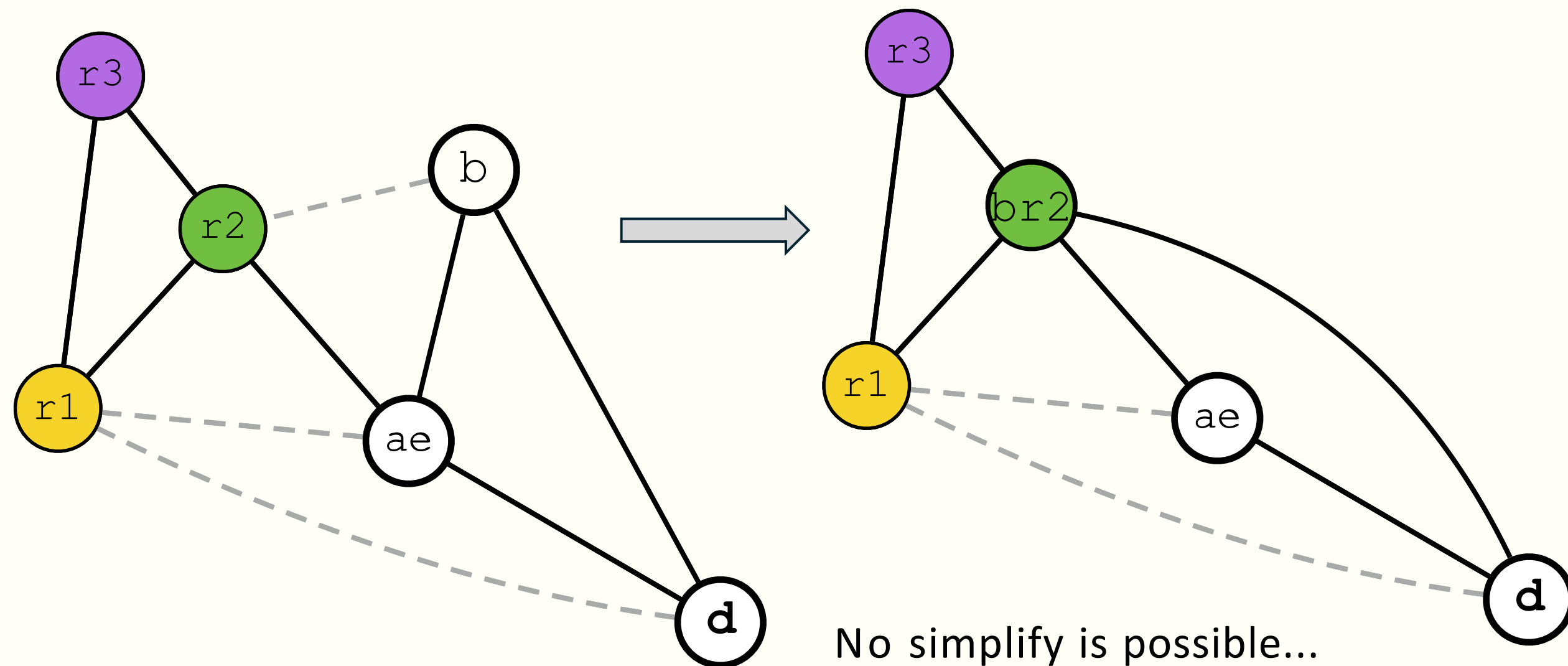
Example: 3 registers (r1, r2, r3)

Stack:
c Spill?



Example: 3 registers (r1, r2, r3)

Stack:
c *Spill?*

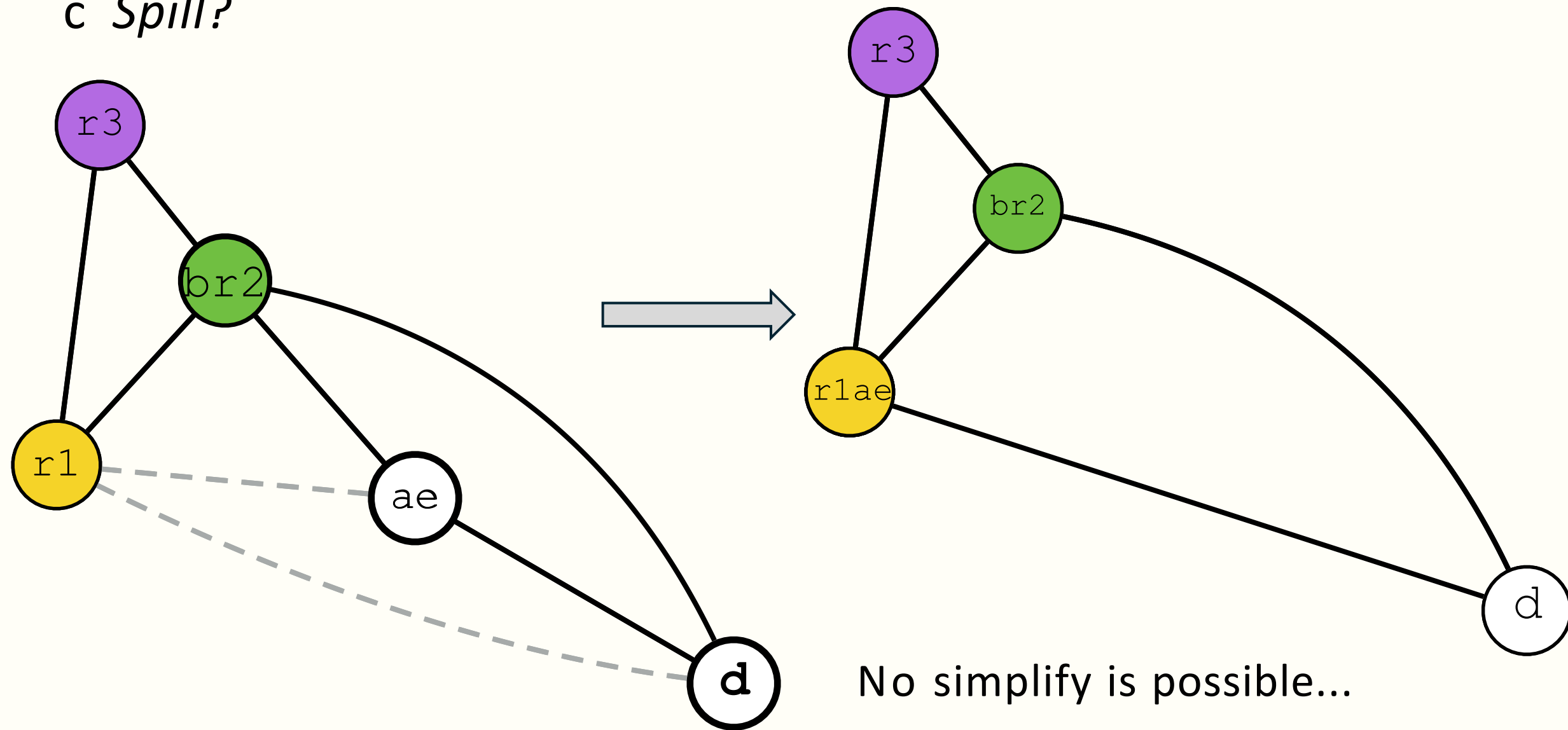


No simplify is possible...
Coalesce `b` and `r2`

Example: 3 registers (r1, r2, r3)

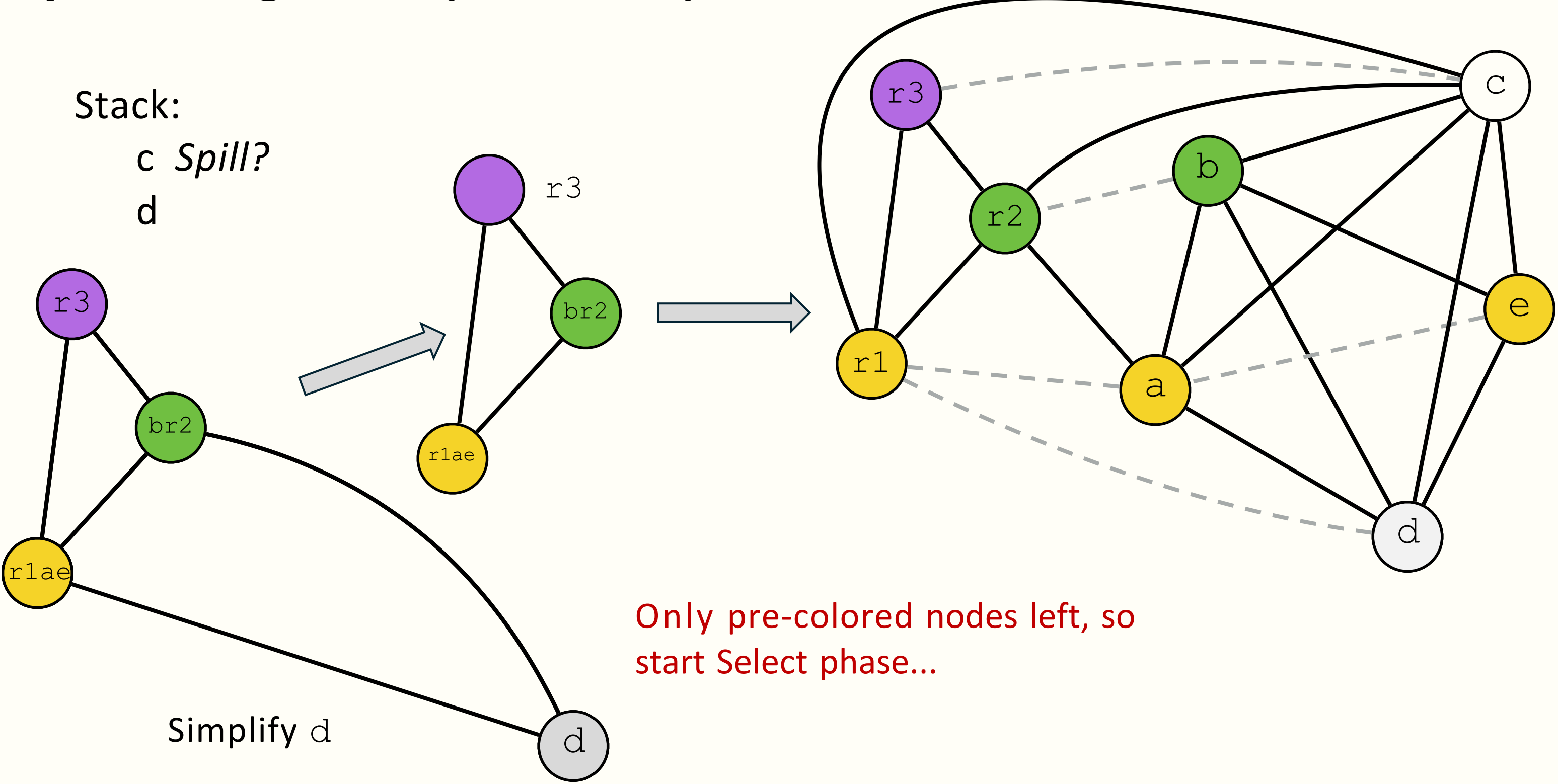
Stack:

c Spill?



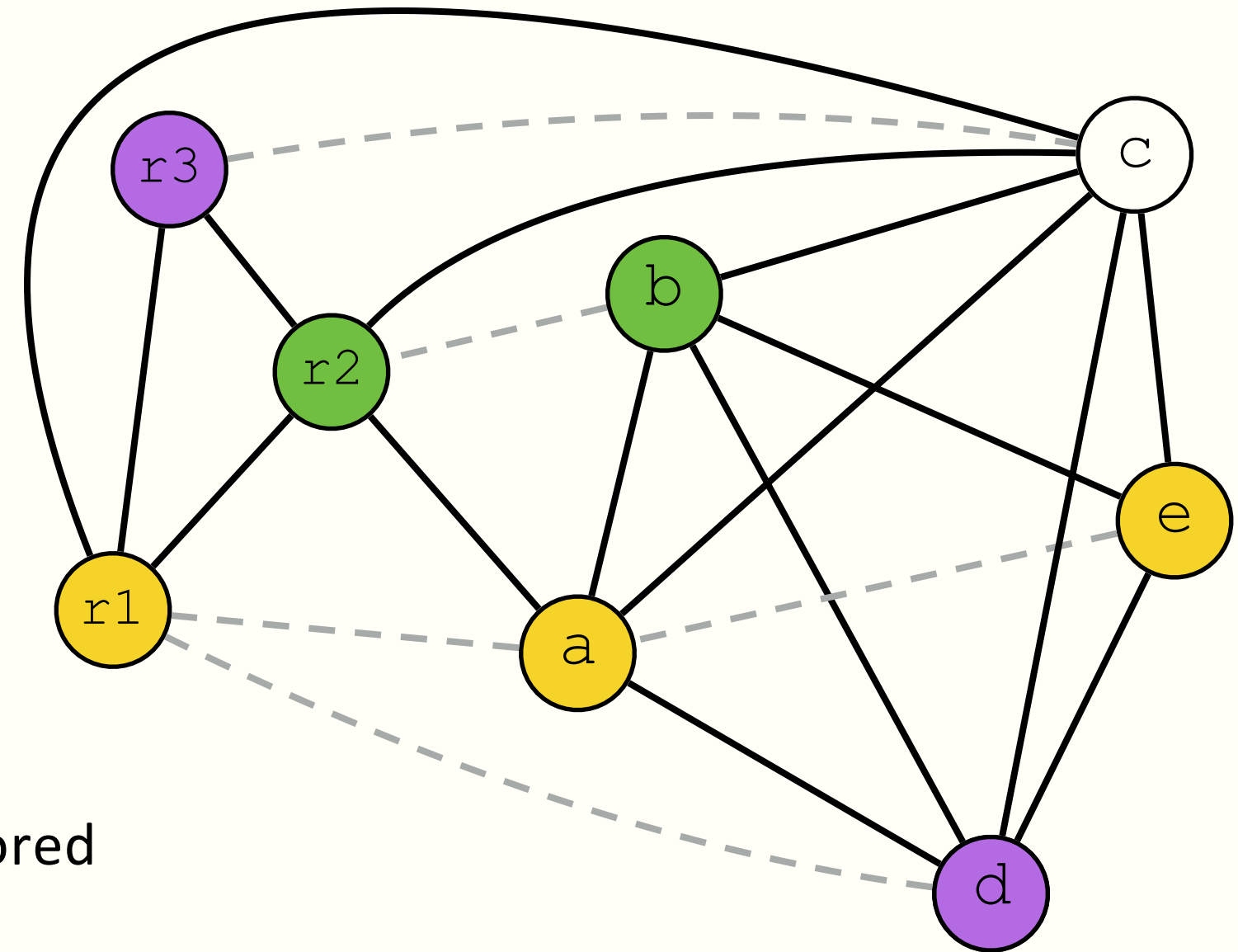
No simplify is possible...
Coalesce `r1` and `ae`

Example: 3 registers (r1, r2, r3)



Example: 3 registers (r1, r2, r3)

Stack:
c Spill?



Due to coalescing, b, a, and e are already colored

Pop d and color it

We can't color c, so we must do an **actual spill**, i.e.,
rewrite code and try again!

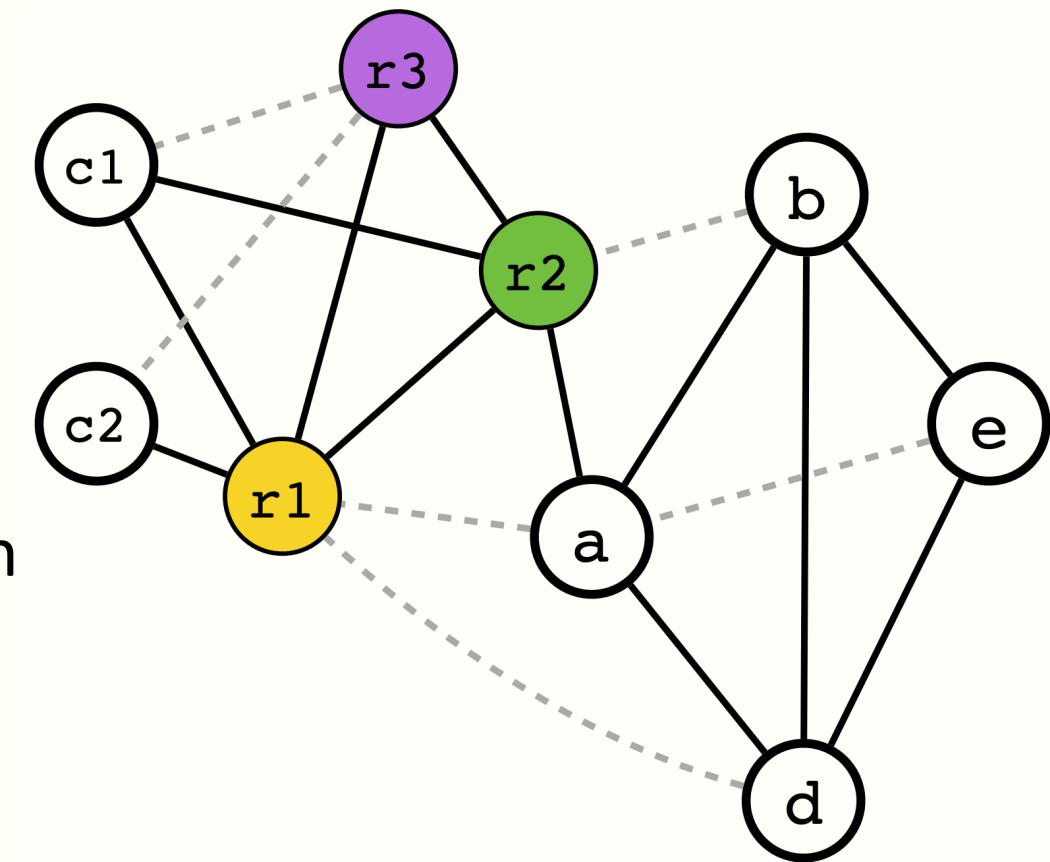
Example: 3 registers (r1, r2, r3)

```
f: c := $r3
  a := $r1
  b := $r2
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 loop else end
end:
  r1 := d
  r3 := c
return
```

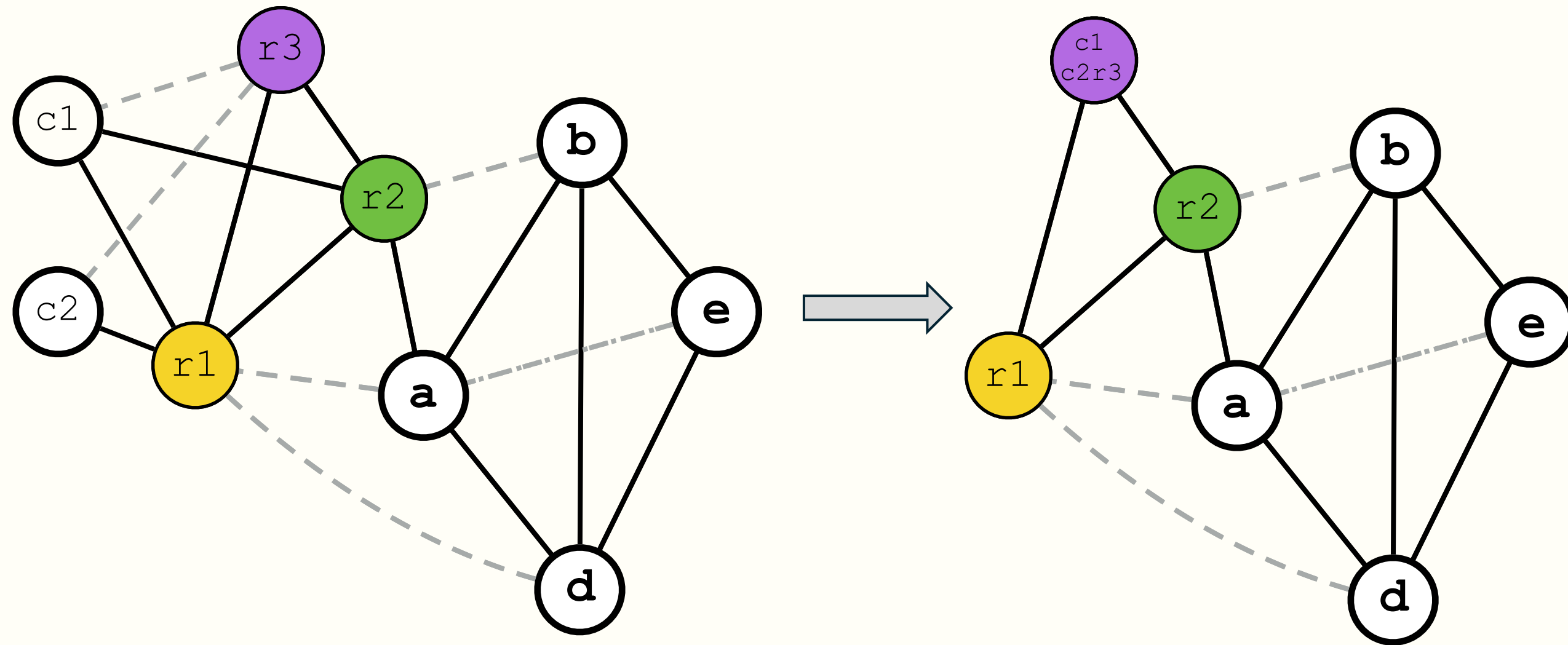
Spill c

```
f: c1 := $r3
  Mem[fp+i] := c1
  a := $r1
  b := $r2
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 loop else end
end:
  r1 := d
  c2 := Mem[fp+i]
  r3 := c2
return
```

Build a new
Inference Graph



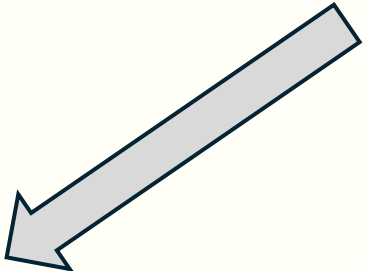
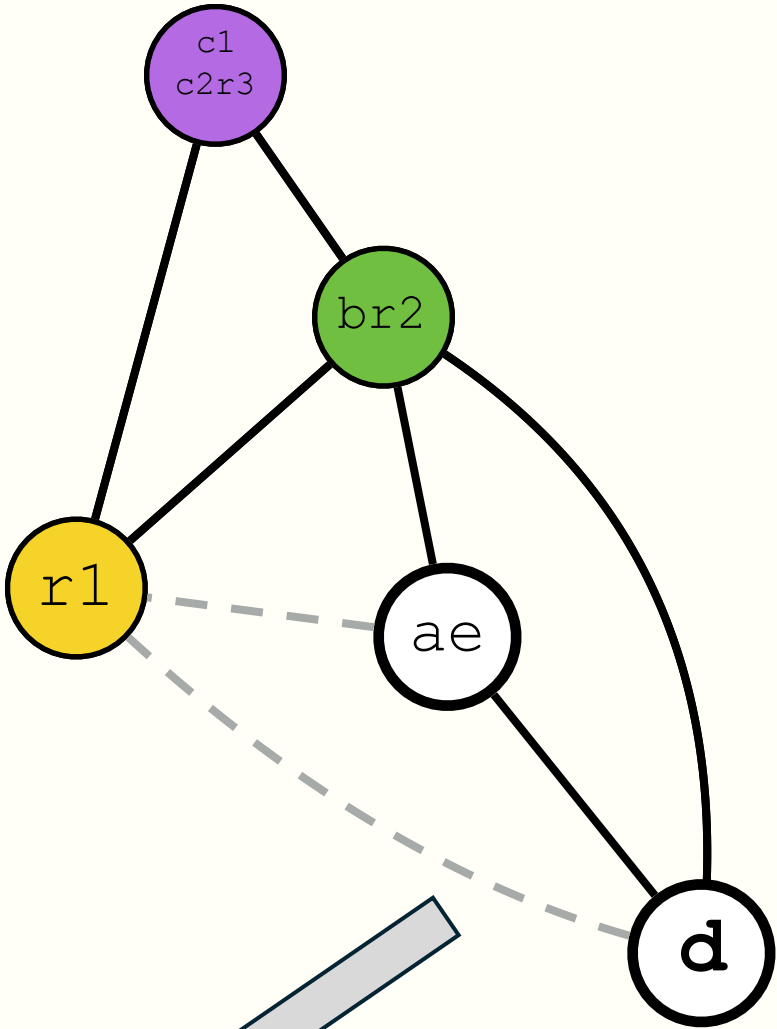
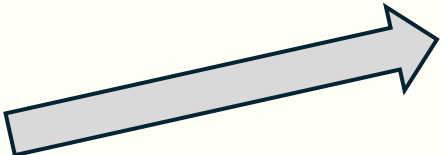
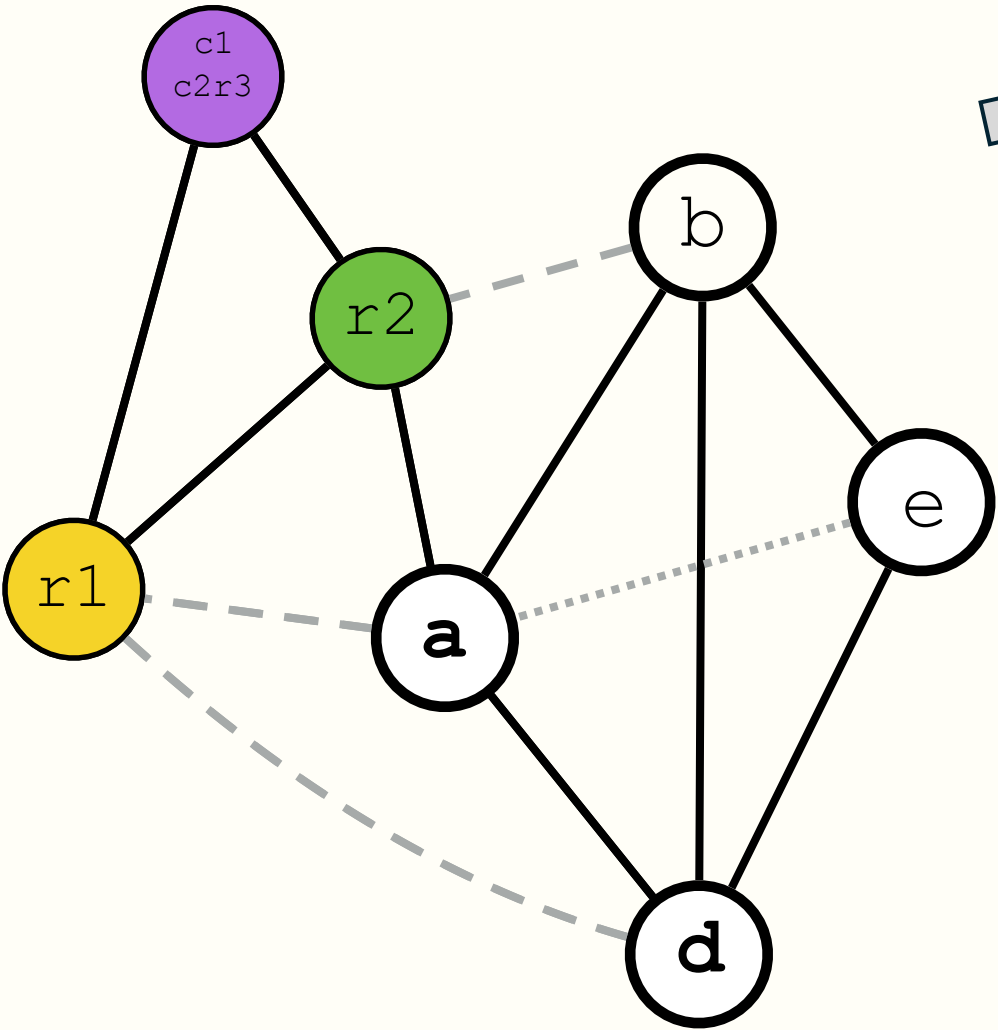
Example: 3 registers (r1, r2, r3)



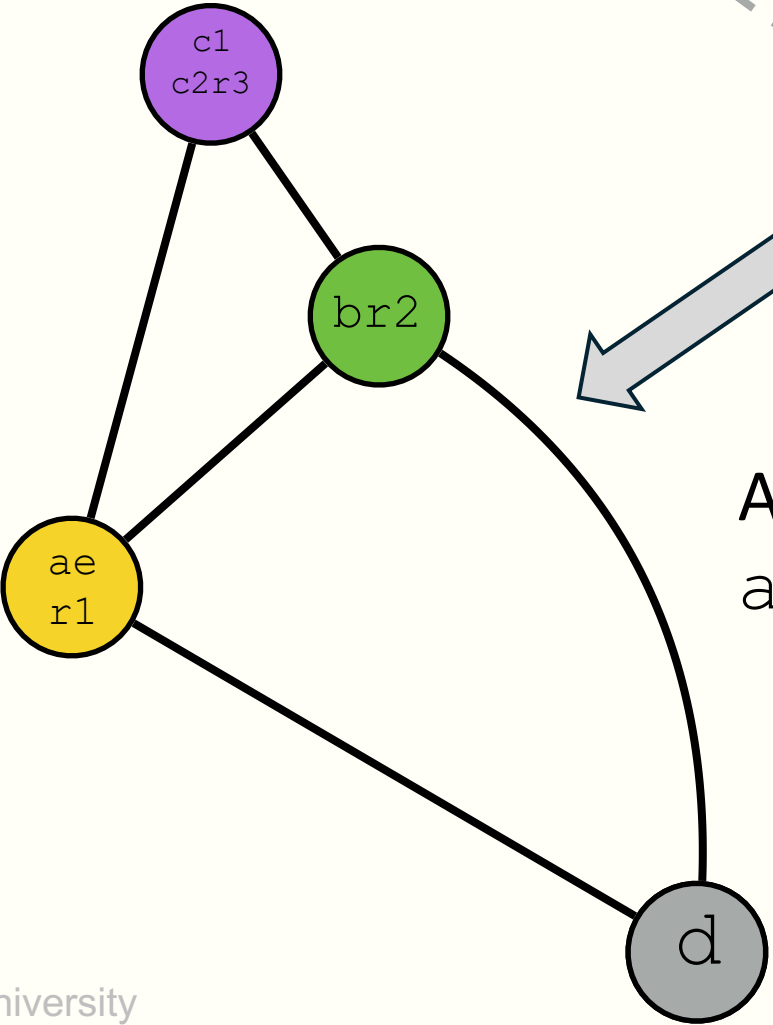
Coalesce c1 and r3, and then c2 and r3

Example: 3 registers (r1, r2, r3)

As before, coalesce a and e,
and then b and r2

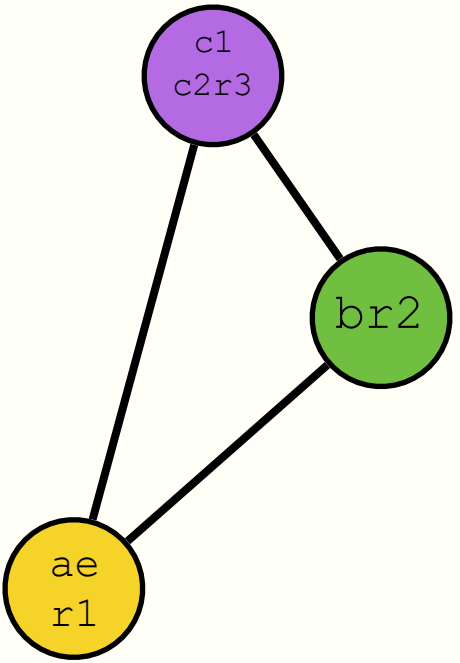
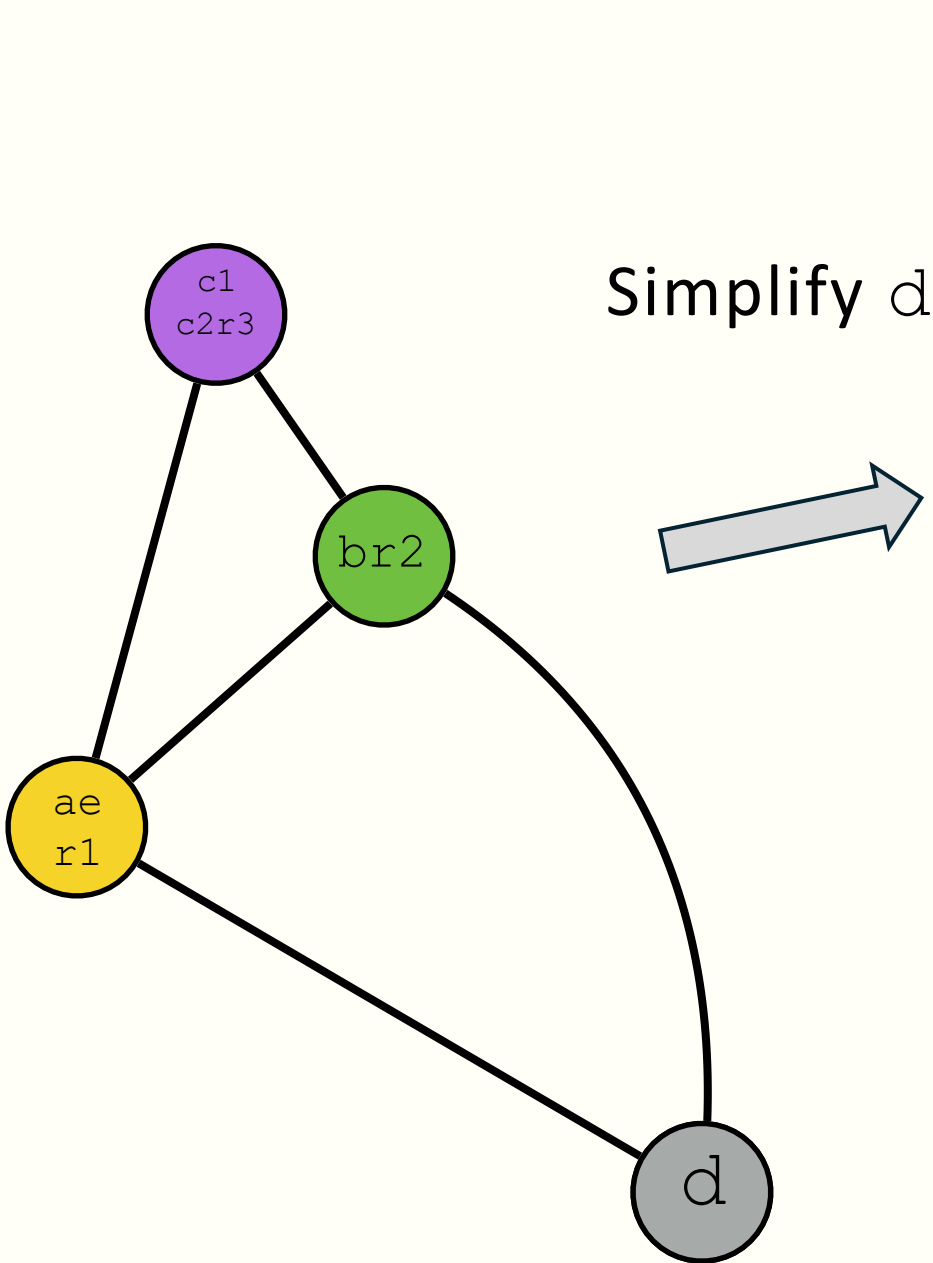


As before, coalesce
ae and r1

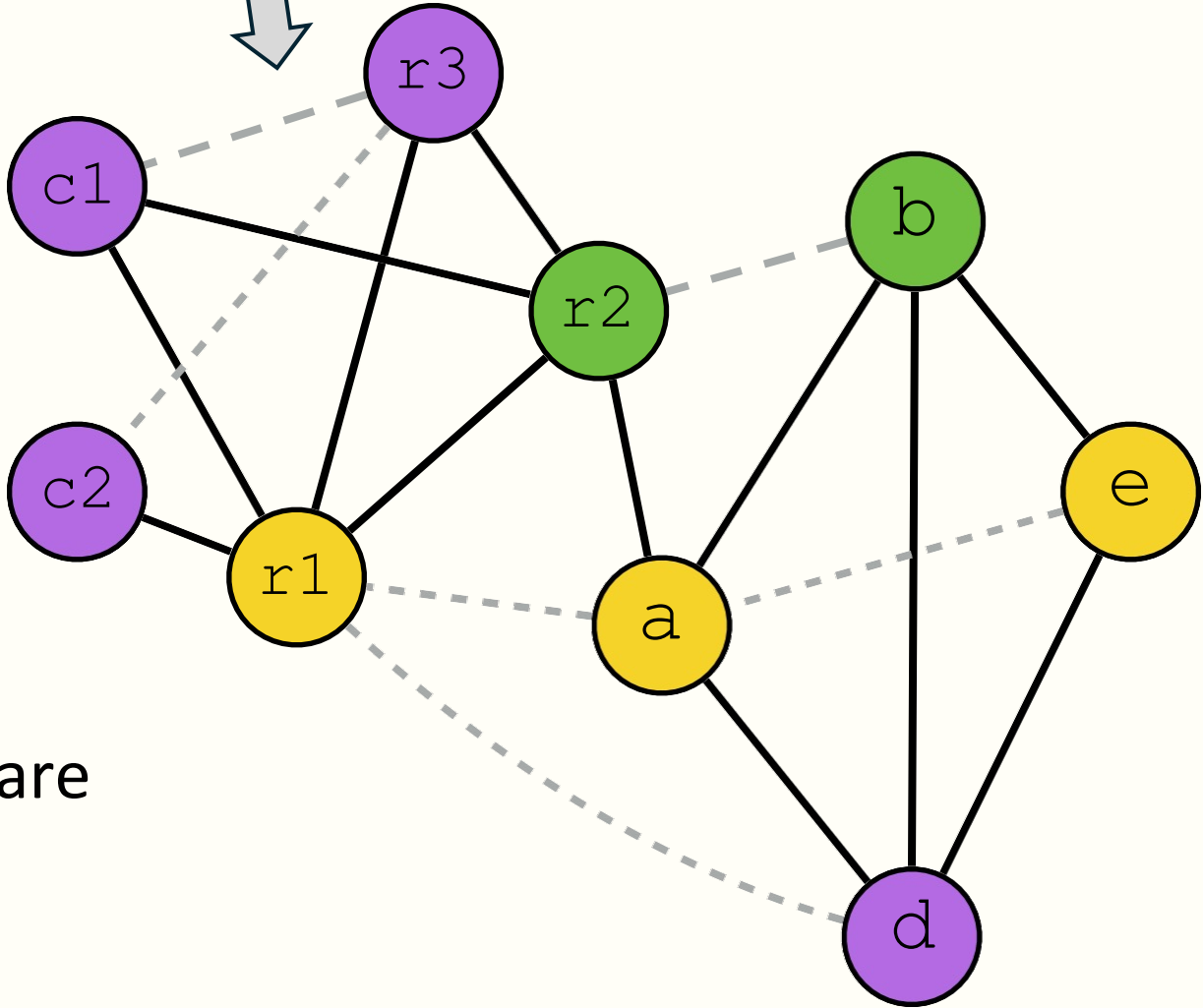


Example: 3 registers (r1, r2, r3)

Stack:
d



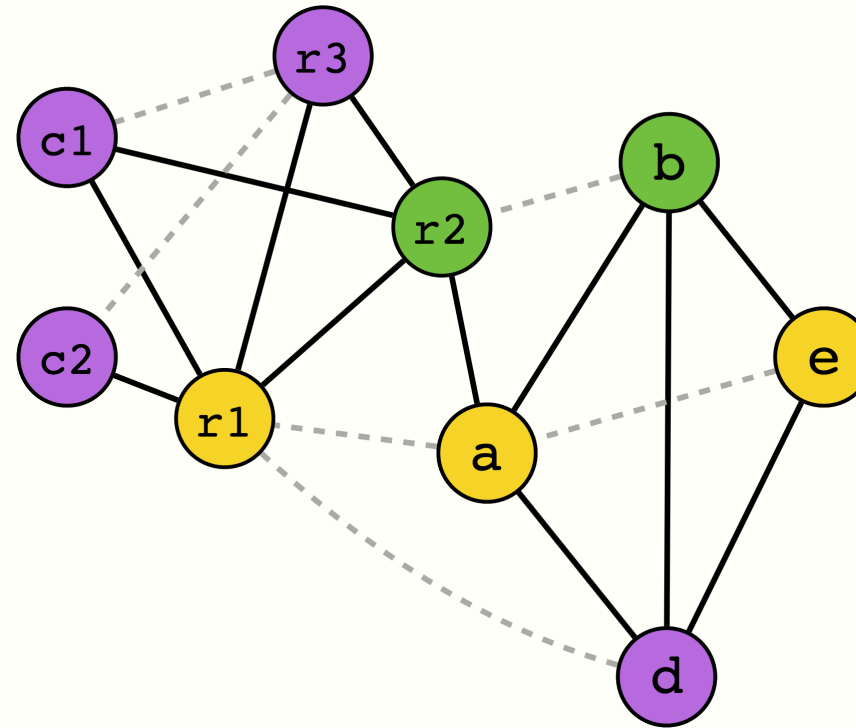
Only pre-colored nodes left, we're ready to move to the **Select** phase!



Due to coalescing, c1, c2, b, a, and e are already colored Pop d and color

Example: 3 registers (r1, r2, r3)

```
f: c1 := $r3
   Mem[bp+i] := c1
   a := $r1
   b := $r2
   d := 0
   e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 loop else end
end:
  r1 := d
  c2 := Mem[bp+i]
  r3 := c2
return
```



Register Allocation

```
f: $r3 := $r3
   Mem[bp+i] := $r3
   $r1 := $r1
   $r2 := $r2
   $r3 := 0
   $r1 := $r1
loop:
  $r3 := $r3 + $r2
  $r1 := $r1 - 1
  if $r1 > 0 loop else end
end:
  $r1 := $r3
  $r3 := Mem[bp+i]
  $r3 := $r3
return
```

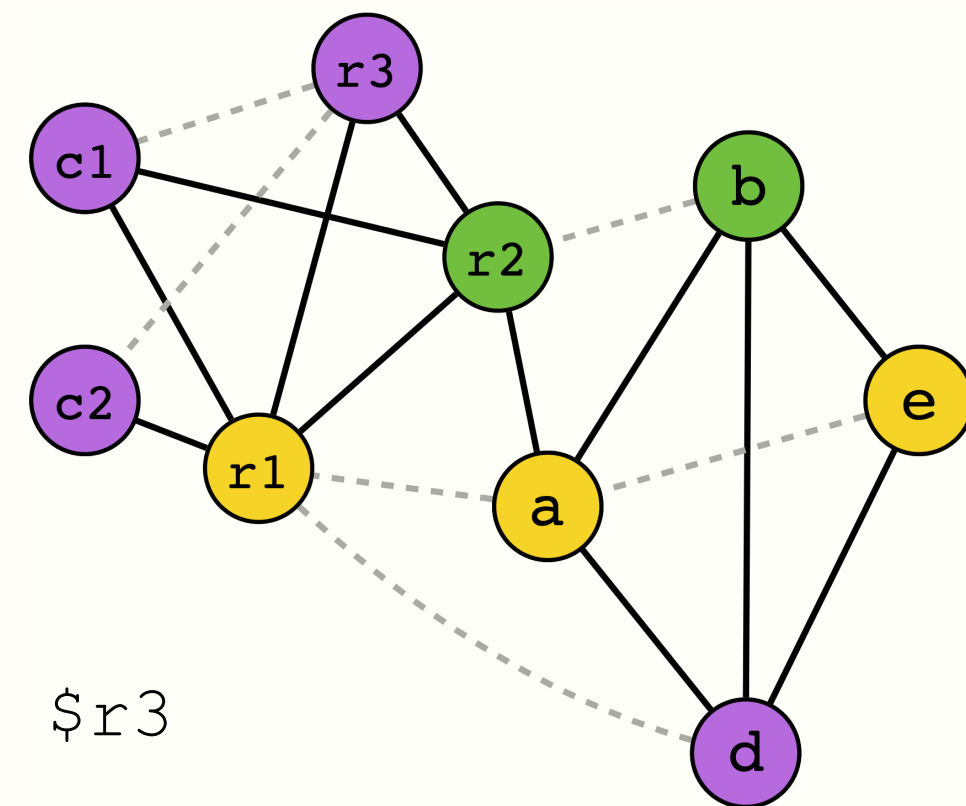
Example: 3 registers (r1, r2, r3)

```
f: $r3 := $r3
   Mem[bp+i] := $r3
   $r1 := $r1
   $r2 := $r2
   $r3 := 0
   $r1 := $r1
loop:
  $r3 := $r3 + $r2
  $r1 := $r1 - 1
  if $r1 > 0 loop else end
end:
  $r1 := $r3
  $r3 := Mem[bp+i]
  $r3 := $r3
return
```

Remove useless Moves



```
f: Mem[bp+i] := $r3
   $r3 := 0
loop:
  $r3 := $r3 + $r2
  $r1 := $r1 - 1
  if $r1 > 0 loop else end
end:
  $r1 := $r3
  $r3 := Mem[bp+i]
return
```



Only one non-coalesced move remains!

Graph Coloring: Our Implementation

- See [GraphColoring.ml](#)
- 支持Simplify和Spill操作
- 暂不支持Coalesce和Freeze操作
 - mv关联的两个虚拟寄存器会尽量染成同一颜色
- 干涉图不支持pre-colored节点
- 可分配寄存器固定为t3-t5, s1-s11
 - t0-t2: spill专用临时寄存器
 - t6: 汇编生成的临时寄存器
 - a0-a7: 传递函数参数
 - 其他: zero, ra, sp, gp, tp, s0/fp
- Spill代价: 静态引用次数/当前度数
 - 未结合循环深度或执行频率等
- 跨函数活跃的寄存器, 只能分配s1-s11寄存器, 不会分配到t寄存器